Various Extensions for the Ambient OSGi framework

Stéphane Frénot Frédéric Le Mouël Julien Ponge Guillaume Salagnac

Université de Lyon, INRIA
INSA-Lyon, F-69621, France
{firstname}.{lastname}@insa-lyon.fr

ABSTRACT

OSGi is a pragmatic wrapper above the Java Virtual Machine that embraces two concepts: the component approach and service-oriented programming. The component approach enables a Java run-time to host many concurrent applications, while the service-oriented programming paradigm allows the decomposition of applications into independent units that are dynamically bound at runtime. Combining component and service-oriented programming greatly simplifies the implementation of highly-adaptive, constantly-evolving applications. OSGi particularly fits ambient requirements and constraints by absorbing and adapting to changes associated with context evolution.

This paper summarizes our INRIA Amazones[1] team work on extending OSGi specifications and implementations to cope with ambient concerns. This paper references our OSGi extension publications divided by three main concerns: management, isolation and security.

1. INTRODUCTION

Using OSGi technology for ambient environments requires focusing on specific problems such as run-time framework size, remote management, remote monitoring and deployment processes.

The reason for this is that ambient intelligence is, and will be, based on hardware which we will refer to in this paper as "gateway devices", that is, middle-sized computing equipments that have much more computing resources that embedded systems like micro-controllers, but much less than tradidional PCs or servers. Examples of these gateways include set-top boxes, mobile phones, automotive embedded systems, or similar equipments. As an illustration, the platforms we used in our experimentations were ARM-based devices as the Linksys NSLU2 (266Mhz CPU, 32MB RAM, 8MB flash) or Sheeva PC plugs (1.2 Ghz CPU, 512MB RAM, 512MB flash). A direct consequence is that deploying and managing OSGi-oriented ambient applications may not be straightforward on those devices, yielding further issues compared to managing "bigger" OSGi-based software stacks like a Java EE server such as GlassFish[2].

In this paper, we compiled most of our current OSGi-related proposals in order to have a synthetic view of the various extensions we investigated. The paper is divided in three sections. Section 2 summarizes the OSGi framework and focuses on the specific concerns of our extensions. Section 3 presents each extension that we provided as a walkthrough our various publications. Finally, section 4 synthesizes our extensions.

[1]`http://amazones.gforge.inria.fr/`

Adamus Workshop in conjunction with ICPS 2010, July 13-15, 2010, Berlin, Germany
Copyright 2010 ACM 978-1-4503-0249-4 ...$10.00.

2. OSGI CONTEXT

OSGi[3] is a container framework built on top of the Java platform. It hosts deployment units called *bundles*, which contain Java resources such as compiled classes, properties files or dynamically linked native libraries. Each bundle features an *Activator* class which is the entry point to be notified when the bundle is started or stopped. A descriptor, expressed as a regular Java manifest, details dependencies between bundles and other meta-data such as the *Activator* qualified name.

The OSGi platform automatically checks dependencies between bundles and controls the lifecycle of each bundle. One key feature of the OSGi framework is the seamless support of application deployment: new applications can be installed, updated and uninstalled at runtime without requiring a restart of the Java virtual machine itself, unlike the large majority of non-OSGi Java applications. This streamlines administration work and improves stability.

Bundles are typically materialized as JAR archives that are fetched by the OSGi framework from a remote HTTP server. Each bundle is associated with a dedicated Java class loader that provides a fine-grained isolation unit with respect to the other bundles. Unlike the typical tree-based classloader hierarchies found in standard Java applications, OSGi provides a directed acyclic graph where each classloader controls which classes and resources should be exposed or not, depending on the bundle manifest meta-data. Bundle dependencies drive the shape of the classloaders graph.

The OSGi underlying programming model focuses on Service-Oriented Programming, or SOP[4]. The driving motivation for SOP is to minimize coupling among software

[2]`http://glassfish.org/`
[3]`http://www.osgi.org/`
[4]`http://openwings.org/`

elements of the application. A service is an interface that describes what a software entity provides in terms of API binding requirements. In turn, many implementations can be provided for the same interface and their details remain hidden to the service requesters. Combining dynamic bundle management and SOP awareness enables a continuous application evolution at run-time.

3. OSGI EXTENSIONS

Since 2004, we provided many extensions to the OSGi framework. Most of them focus on the bundle management layer and aim at providing sound answers to the next three questions :
• How can we improve bundle management and deployment?
• How can we optimize the run-time environment for bundles?
• How can we enforce security for the deployed bundles?

3.1 Bundle Management and Deployment

When dealing with ambient environments we imagine that the gateway is remotely managed. Service providers must be able to install new applications on remote gateways and be able to both manage their behaviors and monitor their execution [21]. In this purpose we provided two extensions: the first is a full implementation of OSGi remote management and the second focuses on an efficient bundle deployment layer.

3.1.1 MOSGi

Management is one of the OSGi specification concerns that was not implemented at the first time. We developed and donated to the Apache Felix[5] community, the MOSGi framework which mainly consists in a wrapper around the JMX management protocol [5]. We provided an end-to-end management layer that covers both the gateway and the client-side part of the architecture. One can find both a run-time for the gateway handling local probes and a run-time manager for the client.

It is worth noting that the manager was also developed as an OSGi bundle and thus provides a really interesting dynamic use-case. When a probe is started on the gateway side, it provides two facets: (i) the MBean facets that provides the management API compliant with JMX protocol, (ii) a specific MOSGi facet that is used by the client side to get a graphical user interface interacting with the remote probe. The client is dynamic: when it connects to the gateway, it requests for the list of installed probes that correspond to the JMX facet integration. Then, for each probe it checks whether it complies with the specific MOSGi facet. If so, it asks the probe for an URL that provides a bundle that brings the graphical user interface. This dynamically discovered graphical bundle is plugged into the manager and starts interacting with the gateway. As an example, the running use-case of the MUSE project [3] was that of a user buying a fridge. When the fridge is installed at the home environment, the set-top box gateway downloads a fridge management probe. When the fridge vendor has to remotely access the fridge, its local manager downloads a graphical user interface that can communicate with the fridge probe and provides a dedicated management user interface.

When we developed the MOSGi framework in a multi-provider environment one issue was to understand the management limits. If any provider can ask for probing information, the gateway can be overloaded by the management layer. If the remote manager asks for the fridge temperature every tenth of a second the gateway can be in some kind of Denial of Service and not be able to do other things. We developed a monitoring scheduling process for gateway [8] through which, we wondered if we could anticipate the increase in CPU load when a new probe is inserted into the gateway. The provided approach is a specific bench generator that measures every installed JMX probe and stores its response time. We show, that the targeted gateway CPU loading curve was regular and that the combination of the various curves leads to good CPU load schedules. If the equipment provider indicates the probe and the pace he wants to query, we can anticipate the CPU load increment. By way of consequences, we can make a distinction between normal behavior and a probe failure since we can observe and anticipate an homogeneous load activity.

3.1.2 POSGi

Peer-to-Peer (P2P) OSGi extension focuses on the efficient and resilient deployment of OSGi bundles. The goal of this system is to substitute the traditional bundle repositories with a P2P architecture. Traditionally, OSGi bundles are installed from a remote web server using an HTTP URL such as `start http://www.somehost.com/abundle.jar`. The server behind `www.somehost.com` may host many bundles and can be overwhelmed by a peak in installation requests. By using a P2P approach, we can avoid this since every gateway can be part of the global, distributed repository. We used Freepastry[6] as a Distributed Hash Table (DHT) overlay network where every gateway and every bundle obtains a specific Freepastry id. DHT yields an efficient routing algorithm between two ids. When sending a new bundle to the network of gateways, it is routed towards the gateway that has the nearest Freepastry id from its own id. When one gateway needs to install a bundle, its request is routed hop-by-hop towards the root gateway that handles the initial archive. On the way back, each gateway on the route can locally store the initial archive. If later, another gateway needs the same bundle, it can be provided either by the root gateway or by any gateway that handles the bundle on the route. POSGi [7] enables URL installation schemes such as `p2p://abundle.jar`. The implementation impacts the OSGi URL scheme handler, adds an alternate local cache where bundles are stored from the Freepastry overlay and a bundle that offers the P2P management.

After having been able to deploy and manage bundles, we focused on the run-time activities of the OSGi framework, so as to optimize bundle execution.

3.2 Run-time extensions

Improving OSGi runtime environments remains a largely open issue. Although it is highly tied to the Java platform, OSGi offered paradigms such as service-oriented programming and bundles bring new modular perspectives on the way Java classes units are to be handled. We separate this section into two concerns. The first one deals with OSGi sandboxing and the second is related to OSGi related byte-code execution optimization.

[5]`http://felix.apache.org/`

[6]`http://freepastry.rice.edu/`

3.2.1 *OSGi sandboxing*

OSGi sandboxing aims at isolating bundle execution and management one from each others. The sandboxing issues we initially investigated were related to the MUSE multi-provider architecture. Indeed, if we host many bundles from various providers on the same gateway, they should be isolated one from each other as they could be in competition. One bundle should not interfere with the one from the other providers. Sandboxing has many levels of granularity on virtual machines: (i) bundle naming isolation, (ii) coding design for proper isolation, and (iii) low-level resources isolation within a virtual machine. We investigated all these three level of isolation.

Virtual OSGi.

VOSGi [22] is a rather naive implementation where we provide OSGi itself as a service. It means that we can assign a specific provider to an OSGi gateway that virtually runs within another OSGi gateway. We provide a start/stop gateway management shell command available from the main gateway, which is in turn called the *core gateway*. In order to interact with each *virtual gateway*, we developed a specific MOSGi probe that enables to remotely manage the virtual instances.

One issue addressed by the VOSGi architecture is the communication between gateways. A virtual gateway may use services provided by another gateway, either the core gateway or another virtual gateway. Our current implementation only enables service exchanges from the core gateway to virtual gateways. When starting a virtual gateway, it declares the service interfaces it wishes to access from the core gateway. At virtual gateway bootstrap time, it receives references to the various corresponding implementations and registers each transmitted implementation within its own registry. Hence, each service exchanged between a core and a virtual gateway is declared in both registries and points to the same implementation.

Nevertheless, one remaining problem in the VOSGi approach is that the OSGi "standard" behavior does not enforces "true" bundle isolation: bundles can obtain as many system resources as they want such as network connexions, threads, and so on. There is no way to constrain a bundle contract within the framework. Moreover, any class can stop the entire virtual machine through a call to `java.lang.System.exit(0)`!. This is a serious issue when hosting multi-provider bundles that may not be "honest". We focused on two developments to improve bundle isolation. The first one is based on a development contract conformance enforcement, while the second one is based on a dedicated virtual machine.

Suspend and Resume [4].

It is an INRIA technical report that raises one simple problem linked to the bundle lifecycle. In the OSGi specification bundles can be in the following states: *Installed*, *Resolved* and *Active*. Although this life cycle seems complete, we believe that it lacks the *Suspended* state. Considering that ambient equipment may be paused and resume and not systematically rebooted, the *Suspended* state is a real ambient concern. At present, the only way of doing this is by managing an internal state when stopping and starting the bundle. It is impossible to distinguish between rebooting and suspending a bundle. One problem raised by introducing the *Suspended* status is that it impacts mostly the run-time model. The *Suspended* status needs to suspend threads, network connexions and all running activities. We can achieve this through two approaches. The next paragraph presents iJVM which is a Java Virtual Machine that aims at constraining resources. The other approach presented in the technical report relies on a specific programming model.

When dealing with the programming model, accessing resources is restricted to some kind of dispatcher. It allocates resources to requesters and maintains a whiteboard associating requesters to resources. When suspension time occurs, the dispatcher knows exactly which resource is associated to each requester, e.g. bundle. In this approach, every direct call to a specific resource allocation should be transferred to a dispatcher that manages identifications. Our implementation adds a specific shell command that integrates the **`suspend #bundleid`** action and a specific **`BundleSuspendableActivator`** interface that adds two methods for suspending and resuming a bundle. Our architecture currently focuses on the threading architecture and only provides a dispatcher to allocate threads and suspend them automatically.

Although this architecture is rather simple to put under operation, it is still dependent on the good will of bundle developers. If they provide standard OSGi bundle they still can be hosted on the gateway, but may present unmanageable behavior. The only workaround for this issue is to constrain bundle behavior at the virtual machine level.

iJVM project.

The iJVM [9] project aims at designing a virtual machine for constrained environment that enforces resource control mechanisms. Even though iJVM had security issues in mind, its main goal is to provide an efficient isolation layer for bundle requesting resources while preserving an efficient communication layer between isolated bundles. iJVM provides three main features for OSGi bundles: (i) memory isolation, (ii) resource accounting and (iii) isolates termination.

VOSGi, Suspend and Resume, and iJVM target run-time isolation for bundle code. The OSGi bundle paradigm offers the appropriate granularity level to enforce isolation. Depending on the desired isolation constraints, each approach offers some level of isolation ranging from a simple naming isolation, to a hard resource constrained environment. But the harder constrains are enforced, the less generic the architecture is.

While trying to enforce isolation and since used hardware have limited resources, we also addressed run-time code optimization.

3.2.2 *Run-time code optimization*

Most of the time bundle life-cycle is driven by the user. In our run-time optimization investigations, we tried to find more automated ways for handling those updates. Two directions were developed for tackling them: the ROCS architecture and the AxSel framework.

ROCS.

The Remotely Provisioned OSGi framework for Ambient Systems [6] project leverages a very simple standard Java principle which is that of remote classes loading. Before starting the activator of a bundle, the OSGi framework performs some activities. It downloads the bundle from a remote location, extracts the jar file on a local cache, extracts the bundle descriptor to control bundle coherence with the current running environment and, provided everything is right, it loads the activator class and invokes its `start` method. ROCS proposes to manage all these activities through a remote approach. The gateway relies on a remote server that hosts the bundles and responds to gateway remote queries. Exploiting the Java remote class loading is straightforward with the OSGi framework. The usage scenario is the following.

When installing a new bundle, it is downloaded from the repository to a remote server and the gateway obtains a local proxy to interact with the remote server. It asks for the bundle manifest, downloads the file from the remote Jar and controls the meta-data to see if every dependency is satisfied. If so, it remotely loads into memory the corresponding `BundleActivator` byte-code and starts the bundle. Then, all subsequent required classes are downloaded on-demand through the `RemoteClassLoader`.

This remote architecture has also been applied to the standard Java runtime classes (e.g., `java.*` and `javax.*` packages) such that it is considered as a plain bundle. We put the necessary classes to bootstrap a minimal OSGi/Java stack into a local classpath classloader. Once started, all remaining standard classes are downloaded from the remote location. Trough our approach, we build an entire OSGi/Java run-time stack that can be hosted on small equipments, less than 8MB of flash, since all loaded classes comes from remote location and are directly brought into memory. ROCS demonstrates that, provided we are connected, we have the same kind of response time as if, we locally cache all bundle classes.

AxSel.

Whereas ROCS focuses on loading classes contained in remote bundles, AxSel [2] aims at optimizing class loading at the service granularity level. The Service-Oriented Programming feature of OSGi specifications enable the description of applications as a composition of services. One application can have implementation variations, depending on the environment. For instance, when using the standard OSGi bundle repository, all dependent bundles are automatically downloaded into the gateway, then started. It maintains a dependency graph from the *Package-Import* and *Package-Export* statements of the bundle manifest. We exploit the *ImportService* manifest property to also maintain a dependency graph between parts of the application. The Axsel framework holds an internal representation of the run-time and regularly polls remote repositories to find new or alternate implementations. When the gateway is in idle mode, it triggers a reconfiguration process, that calculates for every dependent element, bundle or service, if it can find a better implementation. If so, bundles are updated and services are automatically reconfigured.

Axsel and ROCS are complementary approaches: the ROCS system enables a minimal run-time where everything is downloaded on-demand within the main memory, and Axsel brings automatic reconfiguration and optimization of the running environment.

Most of the dynamic features of OSGi framework presented so far, load byte-code into memory from remote locations or from downloaded bundles. We point out many security issues when dealing with isolation. All these concerns led to consider security as specific and generic concern that needs dedicated attention.

3.3 Security around OSGi

Security issues in the OSGi/Java stack are spanning across many points. They arise both at low-level (e.g., bundle code signing) and high-level parts of the OSGi stack (e.g., deployment). The overall OSGi picture reveals many points where security needs enforcements. Our overall approach identified the following elements.

The deployment point of view. Bundle are downloaded from remote locations. Thus, we need some strong guarantees from those locations regarding the bundle origins and integrity. The OSGi specification proposes a signing process that identifies bundle sources and providers, but when we started our study, no implementation was available.

The bundle point of view. Downloaded bundles can lead to two security breaches. Indeed, they can contain code that harms the system, or they can contain code that weakens the framework. We need to consider the two issues to know when bundles are a threat to the framework, and when they open security flaws that could be locally / remotely exploited.

The framework point of view. The OSGi specification does not enforce security constraints as they are mainly implementation issues. Most of current framework implementations are subject to instability if malevolent bundles are installed.

A preliminary work has been conducted to design a bundle signing architecture. This work has been achieved in the context of the MUSE project where we designed a tool-chain to sign bundles and verify their validity at deployment time. The architecture is detailed in [12] and summarized in [14]. An implementation is available with the SFelix[7] project.

After having designed this tool-chain, we worked on a bundle threats catalog aimed at identifying OSGi security weaknesses. The catalog elements are detailed in [13], [17] while a summary is available in [15].

We applied those catalogs to design a hardened OSGi/-Java framework. The [18] article stresses OSGi open-source framework vulnerabilities and expresses guidelines to develop hardened implementations. Yet, some vulnerabilities cannot be controlled at the OSGi framework layer and had to be addressed at the Java Virtual Machine layer. The iJVM [9] virtual machine for isolating threads was mainly designed for this purpose.

The last track we followed when dealing with security concerns was to be able to statically analyze the code of bundles. We developed an install-time analyzer that introspects bundles compiled code to detect vulnerabilities. CBAC [16] describes the proposal. The CBAC control is triggered at the same time as the digital verification signature process, thus only inducing an overhead at deployment time. Unlike the standard Java permissions built-in framework, the CBAC security model does not have any runtime overhead.

[7]`http://sfelix.gforge.inria.fr/`

4. SYNTHESIS

The OSGi extensions that we presented address issues such as management, run-time optimization and security. We summarized our contributions in Table 1.

Extension	Domain	Refs
MOSGi	Management	[21] [5] [8]
POSGi	Management	[7]
VOSGi	Run-time	[22]
Suspend&Resume	Run-time	[4]
iJVM	Run-time	[9]
ROCS	Run-time	[6]
AxSel	Run-time	[2]
SFelix	Security	[12] [14]
Unsecured Bundle Catalog	Security	[13] [17] [15]
Hardened OSGi	Security	[18] [9]
CBAC	Security	[16]

Table 1: Extensions to the OSGi space

Extension	Specification	Patch	Lifecycle	Tool
MOSGi	X	O	O	X
POSGi	O	O	O	X
VOSGi	O	X	O	O
Suspend & Resume	O	X	X	O
iJVM	O	O	O	O
ROCS	O	X	O	O
AxSel	O	O	O	X
SFelix	X	O	X	X
Unsecured Bundle Catalog	O	O	O	X
Hardened OSGi	O	O	O	O
CBAC	O	X	X	O

Table 2: OSGi impacted elements

All our extensions have been developed under the Apache Felix project. Some of them, VOSGi, MOSGi, Unsecured Bundle Catalog have also been validated on Concierge [19]. They all run on standard Intel-based architectures, but many of them, MOSGi, VOSGi, ROCS, SFelix, have also been validated on a NSLU2[8] ARM board, with a dedicated JamVM / GNU Classpath execution stack. All the code that we developed is available from the INRIA GForge project and can be obtained by sending us an email. All code is provided under either the Apache Software License version 2.0[9], or one of the CeCill licences[10].

The provided extensions impact various elements of the OSGi specifications and implementations as summarized in Table 2:

Specification implementation. MOSGi and SFelix propose implementation of OSGi specifications that are not included in the open-source implementations.

Framework Patch. VOSGi, Suspend & Resume, ROCS, Hardened OSGi needs patching some specific version of Apache Felix to operate. Most of the time, the patches correspond to very few lines of code and some extended classes.

Bundle lifecycle modifications. Some of our extensions point out the necessity of extending the standard bundle lifecycle (*Resolved, Installed, Active*) through new status. *Suspended* for Suspend & Resume, *Invalid* for SFelix.

Tools. Tools are elements that do not interfere with the standard OSGi/Java stack. They provide non-intrusive additional services to the framework.

Since iJVM impacts the underlying virtual machine it does not directly concern OSGi framework.

5. CONCLUSION

OSGi is a very interesting environment. Its simplicity and pragmatism are keys to build efficient run-time systems. The INRIA Amazones team focuses on ambient environments and provides extensions that cope with ambient constraints such as limited memory size, remote access, dynamism, context awareness and energy efficiency. Our extensions depend on framework implementations, and as such, they are rather difficult to keep up-to-date with upstream codebase changes. Our current policy is to keep those extensions alive for specific upstream project versions, and to provide upgrades when needed. At present, and due to limited resources within our team, only the MOSGi framework has been donated to the main Apache Felix codebase, under the `mosgi` submodule in the project Subversion repository.

We are still working on various extensions to the framework in different directions. We integrate device mobility concerns in some of our projects and log management features to enhance fault management. However, those projects are still under development and need further validations towards publication.

6. ACKNOWLEDGMENTS

Most of this work would not have been made without PhD students and research engineers. PhD works, mostly in french, of Amira Ben-Hamida [1], Noha Ibrahim [10], Pierre Parrend [11] and Yvan Royon [20] had direct impacts on our extensions and we gratefully thank them all.

[8]`http://gentoo-wiki.com/Gentoo_on_NSLU2`
[9]`http://www.apache.org/licenses/`
[10]`http://www.cecill.info/`

7. REFERENCES

[1] A. Ben Hamida. *AxSeL : un intergiciel pour le déploiement contextuel et autonome de services dans les environnements pervasifs*. PhD thesis, INSA de Lyon, 02 2010. `http://tel.archives-ouvertes.fr/tel-00478169/PDF/these.pdf`.

[2] A. Ben Hamida, F. Le Mouël, S. Frénot, and M. Ben Ahmed. A Graph-based Approach for Contextual Service Loading in Pervasive Environments. In *DOA'2008*, volume 5331 of *LNCS*, pages 589–606, Monterrey Mexique, 2008. Springer Verlag. `http://hal.inria.fr/inria-00395390/en/`.

[3] D'Haeseleer Sam. Detailed requirement-based functional specification of gateway. MUSE IST-6thFP-026442, public deliverable, DB3.1, january 2008. `http://www.ist-muse.eu/Deliverables/WPB3/MUSE_DB3.1p_V1.0.pdf`.

[4] R. Dunklau and S. Frénot. Proposal for a suspend/resume extension to the OSGi specification. Technical Report RR-7060, INRIA, 2009. `http://hal.inria.fr/inria-00423866/PDF/RR-7060.pdf`.

[5] E. Fleury and S. Frénot. Building a JMX management interface inside OSGi. Technical Report RR-5025, INRIA, 2003. `http://hal.archives-ouvertes.fr/inria-00071559/PDF/RR-5025.pdf`.

[6] S. Frénot, N. Ibrahim, F. Le Mouël, A. Ben Hamida, J. Ponge, M. Chantrel, and D. Beras. ROCS: a Remotely Provisioned OSGi Framework for Ambient Systems. In *NOMS 2010*, Osaka Japon, 04 2010. IEEE/IFIP. `http://hal.inria.fr/inria-00436041/PDF/PID1073261.pdf`.

[7] S. Frénot and Y. Royon. Component Deployment Using a Peer-To-Peer Overlay. In *CD'2005*, volume 3798 of *LNCS*, pages 33–36, Grenoble, France, 28-29 November 2005. `http://www.springerlink.com/content/l530048h033r02p8/`.

[8] S. Frénot, Y. Royon, P. Parrend, and D. Beras. Monitoring Scheduling for Home Gateways. In *NOMS'2008*, pages 411–416, Salvador de Bahia Brésil, april 2008. `http://hal.inria.fr/inria-00270941/PDF/sfr_noms_2008.pdf`.

[9] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java Virtual Machine for Component Isolation in OSGi. In *DSN'2009*, pages 544–553, Estoril, Portugal, april 2009. `http://pagesperso-systeme.lip6.fr/Nicolas.Geoffray/files/ijvm.pdf`.

[10] N. Ibrahim. *Spontaneous Integration of Services in Pervasive Environments*. PhD thesis, September 2008. `http://docinsa.insa-lyon.fr/these/2008/ibrahim_n/these.pdf`.

[11] P. Parrend. *Modèles de Sécurité logicielle pour les plates-formes à composants de service (SOP)*. PhD thesis, INSA de Lyon, 12 2008. `http://tel.archives-ouvertes.fr/tel-00362486/PDF/pparrend08phd.pdf`.

[12] P. Parrend and S. Frénot. Secure Component Deployment in the OSGi(tm) Release 4 Platform. Technical Report RT-0323, INRIA, 2006. `http://hal.inria.fr/inria-00084795/PDF/RT-0323.pdf`.

[13] P. Parrend and S. Frénot. Java Components Vulnerabilities - An Experimental Classification Targeted at the OSGi Platform. Research Report RR-6231, INRIA, 2007. `http://hal.inria.fr/inria-00157341/PDF/RR-6231.pdf`.

[14] P. Parrend and S. Frénot. Supporting the Secure Deployment of OSGi Bundles. In *WoWMoM'2007*, pages 1–6, Helsinki Finlande, 2007. `http://hal.inria.fr/inria-00275186/PDF/parrend07adamus.pdf`.

[15] P. Parrend and S. Frénot. Classification of component vulnerabilities in java service oriented programming (sop) platforms. In *CBSE'08*, volume 5282 of *LNCS*, pages 80–96, Berlin, Heidelberg, October 2008. `http://dx.doi.org/10.1007/978-3-540-87891-9_6`.

[16] P. Parrend and S. Frénot. Component-based Access Control: Secure Software Composition through Static Analysis. In *Software Composition*, volume 4954/2008, pages 68–83, Budapest Hongrie, 2008. `http://hal.inria.fr/inria-00270942/PDF/parrend08cbac.pdf`.

[17] P. Parrend and S. Frénot. More Vulnerabilities in the Java/OSGi Platform: A Focus on Bundle Interactions. Research Report RR-6649, INRIA, 2008. `http://hal.inria.fr/inria-00322138/PDF/RR-6649.pdf`.

[18] P. Parrend and S. Frénot. Security Benchmarks of OSGi Platforms: Towards Hardened OSGi. *Journal of Softw. Pract. Exper.*, 39(5):471–499, April 2009. `http://dx.doi.org/10.1002/spe.v39:5`.

[19] J. S. Rellermeyer and G. Alonso. Concierge: a service platform for resource-constrained devices. In *EuroSys '07: Proceedings of the 2007 conference on EuroSys*, pages 245–258, New York, NY, USA, 2007. ACM Press. `http://portal.acm.org/citation.cfm?id=1272998.1273022`.

[20] Y. Royon. *Environnements d'exécution pour passerelles domestiques*. PhD thesis, INSA de Lyon, 12 2007. `http://tel.archives-ouvertes.fr/tel-00271481/PDF/yroyon07phd.pdf`.

[21] Y. Royon and S. Frénot. Multiservice home gateways: Business model, execution environment, management infrastructure. *IEEE Communications Magazine*, 45(10):122–128, October 2007. `http://hal.inria.fr/inria-00270938_v1/`.

[22] Y. Royon, S. Frénot, and F. L. Mouël. Virtualization of Service Gateways in Multi-provider Environments. In *CBSE'2006*, volume 4063 of *LNCS*, pages 385–392, Vasteras, Stockholm, Sweden, June 2006. `http://www.springerlink.com/content/bm003v2666650143/`.