



OSGiTM
Alliance

RFP 156 – Native OSGi

Draft

15 Pages

Abstract

Native OSGi is an effort to revive parts of RFP 89 – Universal OSGi. This RFP specifically focusses on the possibility of implementing an OSGi framework in C and C++. The goal is, to be as close to the OSGi Core specification as possible, without restraining the natural usage of the language itself. Interoperability between C and C++ is important and will receive special attention.

Copyright © Luminis BV 2013.

All company, brand and product names contained within this document may be trademarks that are the sole property of the respective owners.
The above notice must be included on all copies of this document that are made.

0 Document Information

0.1 Table of Contents

0 Document Information.....	2
0.1 Table of Contents.....	2
0.2 Terminology and Document Conventions.....	3
0.3 Revision History.....	3
1 Introduction.....	4
1.1 Rationale.....	5
1.2 Participants.....	5
1.3 Existing Eco-System.....	6
1.4 Technical Background.....	6
1.5 Existing Alternatives.....	7
2 Application Domain.....	7
2.1 Advantages.....	8
2.2 Disadvantages.....	8
2.3 Terminology + Abbreviations.....	8
3 Problem Description.....	9
3.1 C and C++ Language Standard.....	9
3.2 Supported Platforms.....	9
3.3 Memory Management.....	9
3.4 Packaging and bundle format.....	9
3.5 Dependencies and versioning.....	10
Exports And Imports.....	10
Alternative.....	11
3.6 Security Layer.....	11
3.7 Compatibility of C and C++ Bundles.....	12
4 Use Cases.....	12
4.1 Dynamic (Re)Configuration [3].....	12
4.2 Updating and bug-fixing in home automation systems.....	12
4.3 High performance computing software.....	12
4.4 Medical Imaging.....	12
5 Requirements.....	13
5.1 OSGi Compatibility.....	13
5.2 Compatibility of C and C++ Bundles.....	13
5.3 Native language issues.....	13
6 Document Support.....	14

6.1 References.....	14
6.2 Author's Address.....	14
6.3 End of Document.....	15

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 6.1.

Source code is shown in this typeface.

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	24.10.12	<i>Initial document</i> <i>Alexander Broekhuis, Luminis, alexander.broekhuis@luminis.eu</i>
Update	12.03.13	<i>Updates after reviewing the first draft during a OSGi meeting.</i> <ul style="list-style-type: none">• <i>Added additional authors (Sascha and Steffen)</i>• <i>Added overview of current eco-system and frameworks</i>• <i>Detail library usage and platforms</i>• <i>Verify/rewrite requirements</i> <i>Alexander Broekhuis, Luminis, alexander.broekhuis@luminis.eu</i>
Update	16.03.13	<i>nOSGi specific updates.</i> <ul style="list-style-type: none">• <i>nOSGi specific updates</i>• <i>Added related work</i>• <i>Module layer updates</i>• <i>Use Cases</i> <i>Steffen Kächele, University of Ulm, steffen.kaechele@uni-ulm.de</i>

Revision	Date	Comments
Update	22.03.13	<i>Updates:</i> <ul style="list-style-type: none">• CTK Plugin Framework specific updates• Added CppMicroServices• Module layer updates <i>Sascha Zelzer, German Cancer Research Center, s.zelzer@dkfz-heidelberg.de</i>
Update	10.04.13	<i>Comments from Cologne F2F</i> <i>Alexander Broekhuis, Luminis, alexander.broekhuis@luminis.eu</i>
Update	11.04.13	<i>Comments from Cologne F2F</i> <i>Steffen Kächele, University of Ulm, steffen.kaechele@uni-ulm.de</i>
Update	29.08.13	<i>Sascha Zelzer, German Cancer Research Center, s.zelzer@dkfz-heidelberg.de</i>
Update	03.09.13	<i>reworked problem description and requirements, polished document - put focus on requirements</i> <i>Steffen Kächele, University of Ulm, steffen.kaechele@uni-ulm.de</i>

1 Introduction

The OSGi specification currently exclusively focuses on the Java runtime and languages. In the past RFP 89 – Universal OSGi has been started to extend this scope to different languages; C++, C# and ActionScript/JavaScript. Even though RFP 89 is still relevant, it has a very broad scope. This RFP is started to limit the scope to the native languages C and C++. From this point of view, RFP 89 can be seen as an umbrella RFP.

This Document intends to describe the requirements that a native C/C++ OSGi Core specification must meet. This problem is specifically mentioned in RFP-89 as “Native OSGi”. The following description is an update of the content from RFP-89.

“Having a Native OSGi will make it possible to use legacy C/C++ code and more easily than with JNI have it coexist with Java bundles. It would also make it possible to port Java bundles to more efficient C/C++ bundles. It is clear however, that a native OSGi will not support the full set of Java and/or OSGi R5 features (e.g. reflection).”

Even though this section mentions coexistence with Java bundles, this RFP will focus solely on the Core specification. Interoperability with Java bundles can be achieved using e.g. Remote Services.

The scope of this RFP is limited to C and C++ although there are many other languages and compilers that create native binaries. While the focus will be on C and C++, the requirements described in this document should be open enough for these other languages. In other words, requirements not directly related to C and C++ should not force solutions which would block other native languages.

1.1 Rationale

The need for component based development is not new to the native world, several concepts have been introduced to solve these problems (CORBA, COM, etc), but very few have been successful. Over the years OSGi has proven itself to be a stable and accepted solution to this problem, but it focuses specifically on the Java runtime. As of today, many large projects covering all application domains are still written in native languages like C or C++ for various reasons. In certain application domains, modularity and dynamic extensibility are key requirements and being able to use a common native dynamic module system within such projects would be highly beneficial. For these projects it makes sense to be able to use OSGi concepts in a native environment.

For certain areas a binding from Java to C/C++ is enough, but for projects with a greater focus on native languages it makes more sense to have an actual native implementation. For example, if 90% of the codebase is native, it doesn't make any sense to use a Java OSGi framework.

Many of the reasons mentioned in RFP 89 still apply, amongst others:

- Broaden scope of OSGi in other domains
- Ease the introduction of OSGi concepts in non-Java environments

Additionally, there is also re-newed interest in bringing OSGi to other languages besides C and C++, as is demonstrated by the publication of other RFPs like the *Javascript Microservices RFP 159*.

1.2 Participants

Since the initiation of RFP-89 a lot has changed, and several Native OSGi-like implementations have been written. This RFP is an effort of several of these projects with a common goal:

- Combine the effort to reach a greater audience
- Write a common specification which makes it possible to share bundles

At this moment the following projects are participating:

- Apache Felix – A C implementation based on OSGi 4.3 with a focus on Java interoperability and remote services.
- CTK Plugin Framework – A C++ implementation based on OSGi 4.3 using the Qt toolkit. Focuses on adhering to the Core API as close as possible.
- NOStrum (formally called nOSGi [5]) – A C++ implementation based on OSGi 4.* with a unique bundle wiring mechanism and a focus on embedded devices.
- CppMicroServices – A C++ implementation based on the ideas of “OSGi Lite”, i.e. without the dynamics of the module layer. Its focus is on a type-safe service layer API in C++.

1.3 Existing Eco-System

These participating projects already provide a fairly extensive collection of services and bundles. The expectation is that these bundles will be adapted to be compatible with the Native-OSGi specification. This provides an extensive eco-system that can and will be leveraged by the Use-Cases described in Chapter 4.

The following overview lists all the bundles/services provided by the different participating projects:

- Apache Celix
 - OSGi Compendium Chapter 101 – Log Service
 - OSGi Compendium Chapter 103 – Device Access
 - OSGi Enterprise Chapter 122 – Remote Service Admin
 - Log Writer
 - Shell Service
 - Shell Textual User Interface (TUI)
 - Dependency Manager (based on the Apache Felix Dependency Manager)
- CTK Plugin Framework
 - OSGi Compendium Chapter 101 – Log Service
 - OSGi Compendium Chapter 104 – Configuration Admin
 - OSGi Compendium Chapter 105 – MetaType Service
 - OSGi Compendium Chapter 113 – Event Admin
- NOStrum (nOSGi)
 - Support for Module Layer
 - Shell Service
 - Remote Shell Service
 - Job Scheduling Service (cron bundle)
 - Bus Access Service (1-wire bundle)

1.4 Technical Background

The participating projects already provide partial implementations of the concepts described in the OSGi Core specifications. Each implementation has different strengths and may differ from the others in the choice of solutions and techniques for implementing certain Core features. Within Native-OSGi the experience gained while

developing these solutions and techniques will be used to create a common OSGi Core specification for the C and C++ languages.

In the following, distinct features of each OSGi-like implementation existing today and being part of the Native-OSGi efforts are listed:

- Apache Celix uses a ZIP-based bundle format and provides a C API covering large parts of the Core specifications.
- NOStrum (nOSGi [4]) provides a module layer implementation with a wiring model for dynamic updates of versioned shared libraries, based on runtime ELF header patching.
- CTK Plugin Framework provides an almost complete C++ API for the module, life-cycle, and service layer including support for bundle resources.
- CppMicroServices provides a C++ service layer implementation with a focus on type-safety.

Combining these efforts and taking advantage of the already acquired expertise is assumed to be highly beneficial for the development of a Native-OSGi specification.

1.5 Existing Alternatives

Alternatives to Native OSGi are:

- CORBA and the CORBA Component Model
- Service Component Architecture (SCA)
- Open API Platform (OpenPlug) from Alcatel-Lucent

The CORBA component model proposes an alternatives to OSGi. It was designed to support composition and deployment of applications on distributed resources. The component standard defines the interfaces (ports), dependencies, and rules managing interaction between components as well as their lifecycle. Yet, the tight coupling between code modules with respect to dependencies hinders flexibility and adaptability of applications.

Lifecycle management of Open-Plug is quite similar to the one proposed in Native OSGi. However, to run code on devices, it uses proprietary technology that interprets code. A Native OSGi implementation should use open standards and execute code natively on devices. Moreover, Native OSGi uses standard method invocation for inter-component communication, whereas Open-Plug uses a more heavyweight message bus.

2 Application Domain

Native OSGi can be applied in any domain which has/wants to rely on native code. Common examples are:

- Embedded software vendors
- (Medical) Imaging solutions
- Sensor Networks

Since this RFP focuses on a C/C++ OSGi Core Specification, it is difficult to list the final application domain. A few usage examples are:

- A software vendor has a large distributed (embedded) software stack, and wants to be able to dynamically (re)configure a running system. For this they need a module based system which has the ability to replace a module at runtime.
- A software vendor has a mixed software base of Java and C/C++ software. The Java software already uses OSGi. To be able to leverage the same benefits, they want to have a similar solution for their native code.

2.1 Advantages

Compared to the traditional way of writing shared libraries, a Native OSGi system has the following advantages:

1. Service – oriented modularity concepts for native developers, benefiting from years of experience gathered within the Java OSGi community.
2. Dynamic updates and reconfigurability of native code in a standardized way.
3. Alternative to JNI for Java and native code interoperability.

2.2 Disadvantages

1. C/C++ lacks development tools when compared to Java, which could make the usage of OSGi concepts difficult for the average programmer.
2. The lack of language features like reflection will limit the scope of a Native OSGi specification.

2.3 Terminology + Abbreviations

Native OSGi: Working title for a C/C++ OSGi specification.

N-OSGi: Shorthand for Native OSGi.

Shared Library: This document consistently uses the term "shared library" for code loadable at runtime. On Windows, they are called Dynamic Link Libraries (DLL), on UNIX systems Dynamic Shared Objects (DSO), and on Mac OS platforms Dynamic Libraries (DyLib).

Loader: A platform-specific program responsible for loading shared libraries into memory and resolving their dependencies.

Platform: A combination of processor architecture, operating system, shared library format, and compiler.

3 Problem Description

The Native OSGi specifications are assumed to be mostly written in a platform-independent way. They should refer to the supported C and C++ language standard, with the exception of specifying the resolving process. The resolving process will likely need to take platform-specific and/or object file format features into account.

3.1 C and C++ Language Standard

As every language, the C and C++ languages evolve over time. A Native OSGi specification requires a definition of a language standard. Due to the large amount of different compilers and platforms, a suitable language version is required to not restrict the usage of Native OSGi too much.

3.2 Supported Platforms

A platform for native systems typically is a combination of processor architecture, operating system, and compiler. While the JVM provides a standardized runtime platform for OSGi, this will not be the case for Native OSGi. Native OSGi requires a minimum set of platforms on which compliant implementations are required to run, similar to the concept of Java *execution environments*.

3.3 Memory Management

Java relies on a garbage collector to reclaim memory from unused objects whereas C/C++ does not impose any kind of memory management system. Especially with regard to service objects, memory management of these objects is an important issue. It is also necessary to specify the life-time and ownership of all objects specified in Native OSGi.

3.4 Packaging and bundle format

OSGi ships bundles in the JAR format. However, JAR is explicitly bound to Java. Native OSGi has to specify an comparable ZIP-like bundle format.

As bundle contain native code, they are bound to a specific platform. Native OSGi bundles thus require meta data describing the platform architecture and needed runtime environment. Platform-independent bundles require source code. Native OSGi has to specify how build such bundles.

In the native world, libraries (e.g. ELF format on Linux, Mach-O format on MacOSX, and PE format on Windows) are a standard way to split applications. Yet, there are multiple ways to package code into libraries. Libraries can exist on bundle, package and class level, but libraries packages level have serious advantages:

- Have multiple libraries per bundle
- Have exported and private libraries

Just like Java Packages this makes it possible to make a distinction between the private implementation and the exported services.

- Allow code sharing

Using exported libraries it is possible to share code between bundles.

Another issue is the search path of libraries. A Native OSGi solution has to set up the search path in a way that it can fully control wiring of libraries and bundles.

3.5 Dependencies and versioning

The Module Layer of the OSGi Core specification defines several dynamical code loading features, but relies on Java classloading mechanisms. The nOSGi project already presents solutions for many native specific issues. A solution has to take care of loading and wiring the different libraries.

An important aspect of OSGi is versioning of packages, and the wiring needed to find the correct version of a dependency. Whereas Java by itself does not provide any default versioning mechanism, the different native linkers do. These versioning mechanism usually allow to load and access code libraries in different version at the same time. Unix uses a versioning scheme for shared libraries with a triple major.minor.micro. By convention this versioning scheme is included in library names. MacOS mach-o file format also provides version information comparable to Unix. The windows Portable Executable format also has version information. Yet, it not used during loading the DLL. Native-OSGi should use an abstract version scheme that is compatible with popular platforms.

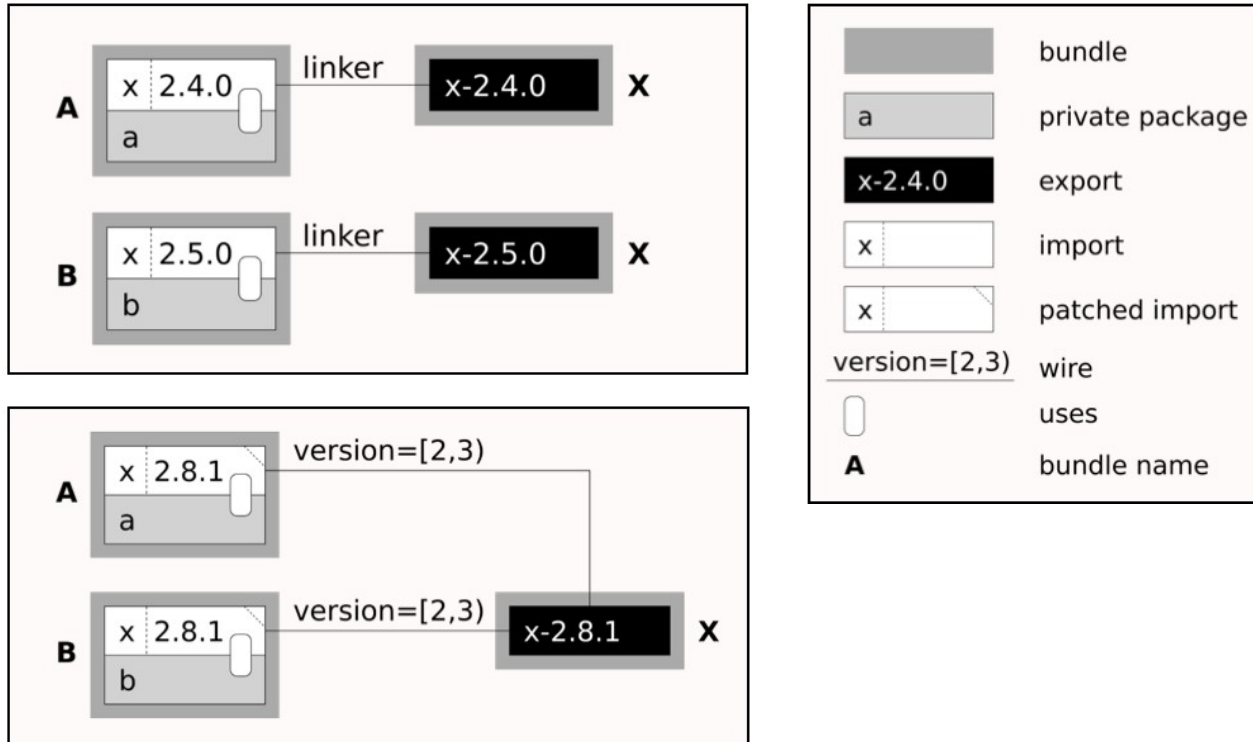
To establish a dynamic wiring, libraries and its versions should be altered to match dependencies of OSGi wiring at runtime (see nOSGi). It is essential to allow multiple versions and dynamic bundle updates.

Mention Application Binary Interfaces (ABIs), their compatibility, and the relation to versioning schemes.

Exports And Imports

The following diagram depicts the wiring process. In the upper diagram the situation at link-time is detailed. The library dependencies are hard-coded by the linker during the linking process. This results in a situation where Library A depends on version 2.4.0 of Library X, while Library B, developed with a different set-up, depends on version 2.5.0 of Library X.

If in the Meta-Data of the bundle providing library A or B the version range [2, 3) is provided, the module layer (resolver) looks for any bundles which fit in this range. Since the linker already has embedded the version info in the library itself, this information has to be updated. So the version of the providing bundle/library (found by the resolver) is used to patch the version required by library A and B.



In a Native OSGi setup, interoperability may be reduced when using compilers of different vendors. As a consequence, only bundles that were compiled with a compiler of the same vendor may share code. In most cases, bundles of the system are compiled with one certain compiler. Using an intermediate format such as LLVM or source bundles may allow to mix bundles that were compiled with different compilers (see [4]).

Alternative

Even though the model depicted above works, it might be worthwhile to investigate different solutions. A possible alternative is to extend the used linker (or linkers) to solve this. This would mean that instead of linking to a fixed version, the version range can be used. In this case there is no need to update the libraries at runtime. However, this might introduce restrictions to the target environment, e.g. a proper linker has to be supplied for every operating system, runtime environment and possible versions.

A much more complex solution might be the usage of one dedicated linker on all platforms. This linker can be bundled together with the framework which means the platform linker is ignored. Benefits of this are:

- No different library format, which means no different solutions (bundles or anything else) for different platforms.
- Only need for one common solution for wiring.

3.6 Security Layer

Isolation in native code is not a trivial task. There is no integrated security manager that can limit access to resources for parts of application code. Using pointers memory can be arbitrarily modified.

3.7 Compatibility of C and C++ Bundles

For both programming languages, C and C++ own framework support is necessary. The way of designing API which directly affect how to access services differs in these languages. Thus, Native OSGi has to specify a way to call a C service via a C++ interface and vice versa.

4 Use Cases

4.1 Dynamic (Re)Configuration [3]

A software vendor has a large distributed (embedded) software stack used for a sensor network. To be able to react to a changing environment it must be possible to reconfigure software. Since this can happen during actual usage this must be done dynamically and with little impact on the running network.

To be able to do this, a module based architecture is defined. This architecture will be realized using Native-OSGi. OSGi provides a clear component based model, as well as all aspects needed to be able to reconfigure the system at runtime (through the module and life cycle layer).

4.2 Updating and bug-fixing in home automation systems

A number of software modules for a security system or an home gateway router have been developed. The system should provide support for dynamical updates e.g. in term of bug-fixing or to add new functionality. End-users should further be able to extend their system with additional software (i.e. plugins) with no need to restart the entire platform. To keep the hardware costs low, software is typically developed in native languages and for exotic hardware architectures such as Mipsel. A native OSGi implementation can bring the dynamical aspects to these devices without increasing the Bill of Material.

4.3 High performance computing software

A developer wants to extend mathematical software with a new algorithm. He is able to implement his algorithm according to an existing interface, and can deploy his algorithm to the OSGi-based existing software.

4.4 Medical Imaging

A medical imaging related research group or company wants to create an application which integrates different components to achieve a complex work-flow. This work-flow includes preprocessing of image data on a server, the retrieval of the preprocessed data from the server, the loading of the data into memory, suitable visualization of the data, specific user interaction, followed by the calculation of some parameters and storing them in a database. Because the computational complexity of some of these components is very high and the execution time of the complete work-flow is critical (cost factor), the components are written in C++ and highly optimized. Ideally, each component would communicated with the others through well-defined interfaces implemented as OSGi services to increase their reusability and lower update and maintenance efforts for deployed systems.

5 Requirements

5.1 OSGi Compatibility

- OSGI-1 Native OSGi MUST detail how the Module Layer should be supported. This support SHOULD be based on the module layer described in the chosen specification on [5].
- OSGI-2 As the bundle format will include specific Native OSGi headers, the solution MUST use a specific header prefix.
- OSGI-3 The solution MUST use a well defined, ZIP-like platform independent format for bundles similar to the format described in the chosen specification on [5].
- OSGI-4 Native OSGi MUST define how to build platform independent bundles (i.e. source bundles)
- OSGI-5 The solution MUST follow the life cycle for bundles as defined in the specification on [5].
- OSGI-6 The solution MUST detail a service layer and registry similar/equal to the chosen specification. This support SHOULD be based on the service layer described in the chosen specification on [5].
- OSGI-7 The module layer MUST support multiple libraries even of the same name but in different versions.
- OSGI-8 The module layer MUST support export control of libraries through the use of export/import headers in the bundle manifest. It MUST support exported and private libraries.
- OSGI-9 The module layer MUST support versioning using export versions and import version ranges.

5.2 Compatibility of C and C++ Bundles

- CPPC-1 The solution SHOULD support C and C++. This support has to be in language natural matter. C++ developers should be able to use C++ constructs, whereas C developers should be able to do the same using C constructs.
- CPPC-2 The solution SHOULD support consumption of C services in a C++ Framework implementation and vice versa. Services SHOULD be solved in a transparent way without any additional work on the consuming end.
- CPPC-3 The service registry SHOULD be able to handle C as well as C++ services.

5.3 Native language issues

- LANG-1 Native OSGi MUST define a language standard. Due to the large amount of different compilers and platforms, a suitable language version must be chosen to not restrict the usage of Native OSGi too much.

- LANG-2 The Native OSGi specification **MUST** state a minimum set of platforms on which compliant implementations are required to run, similar to the concept of Java *execution environments*.
- LANG-3 The solution **MUST NOT** rely on functions of specific operating systems **NOR** alter the standard system environment (e.g. standard runtime libraries)
- LANG-4 Native OSGi **MUST** address memory management of all object specified in Native OSGi, in particular with regard to service objects. This covers both, the life-time and ownership of objects.
- LANG-5 The solution **SHOULD** introduce no or little overhead when issuing calls between bundles. Where possible wrappers must be avoided. The case of C – C++ interaction will be an exception to this requirement.
- LANG-6 The Module Layer **MUST** take care of loading and wiring the different libraries. Moreover, it should define an appropriate bundle format.
- LANG-7 Native OSGi **SHOULD** use shared libraries on the level of Java Packages.
- LANG-8 Native OSGi **SHOULD** use a version scheme that is compatible with library version schemes of popular platforms.

6 Document Support

6.1 References

- [1] Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.
- [2] Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0
- [3] Sensor Technology Applied in Reconfigurable Systems, <http://starsproject.nl/>
- [4] S. Kächele, J. Domaschka, H. Schmidt, and F.J. Hauck, nOSGi: a posix-compliant native OSGi framework. In Proceedings of COMSWARE. 2011, 4-4.
- [5] OSGi Alliance Core Specifications, <http://www.osgi.org/Specifications/HomePage>

6.2 Author's Address

Name	Alexander Broekhuis
Company	Luminis BV
Address	IJsselburcht 3 6825 BS Arnhem
Voice	+31 622393303
e-mail	alexander.broekhuis@luminis.eu

Name	Steffen Kächele
Company	University of Ulm
Address	Albert-Einstein-Allee 11 89081 Ulm
Voice	
e-mail	steffen.kaechele@uni-ulm.de

Name	Sascha Zelzer
Company	German Cancer Research Center
Address	Im Neuenheimer Feld 280 69120 Heidelberg
Voice	
e-mail	s.zelzer@dkfz-heidelberg.de

6.3 End of Document