An OSGi Implementation and Experience Report

Richard S. Hall and Humberto Cervantes
Laboratoire LSR IMAG, 220 rue de la Chimie
Domain Universitaire, BP 53, 38041
Grenoble, Cedex 9 FRANCE
{Richard.Hall, Humberto.Cervantes}@imag.fr

Abstract – **The Open Services Gateway Initiative (OSGi) defines and promotes open specifications for the delivery of managed services into networked environments. A key element of this initiative is the OSGi framework, which is a lightweight framework for deploying and executing service-oriented applications. This paper focuses on the OSGi framework by first discussing some implementation details of an open source implementation of the OSGi framework, called Oscar. The paper then presents issues that arose or whose importance was magnified through implementing and/or using the OSGi framework.**

I. INTRODUCTION

The Open Services Gateway Initiative (OSGi) is an independent, non-profit corporation working to define and promote open specifications for the delivery of managed services to networked environments, such as homes and automobiles [11]. The initial market for OSGi was home services gateways, where the vision was a future in which a house would contain a home-area network and most, if not all, household devices would be connected to this network. The home services gateway would then act as a gateway between the end user (and owner) of the devices and the service providers that want to provide (i.e., sell) services for the devices. Prime examples of this vision are home security and home health care monitoring. This vision was further expanded to include any networked environment, such as automobiles, that provided a mix of embedded devices and the need to deploy services for those devices.

To achieve this vision, OSGi defined a specification for the OSGi Service Platform [10]. The OSGi Service Platform consists of two pieces: the OSGi framework and a set of standard service definitions. The OSGi framework defines the actual services gateway (i.e., the deployment and execution environment for services) and is the focus of this paper. The OSGi framework is a lightweight framework for deploying and executing service-oriented applications. It provides a simple component model, a service registry, and support for deployment. Because of these characteristics, the OSGi framework can also be used for purposes other than services gateways.

Research work in using the OSGi framework as the basis for creating service-oriented, component-based applications is ongoing in the Gravity project [5]. Other uses include multi-agent systems and plugin mechanisms. In particular, the Eclipse project [6], a modular, plugin-centric integrated development environment spearheaded by IBM, is experimenting in using OSGi as a dynamic plugin mechanism; this experimentation is taking place in an Eclipse subproject called Equinox.

This paper is relevant to both views of the OSGi framework, either as a services gateway or as a generic application framework. The paper first describes the OSGi framework in more detail. After this detailed description, the paper presents some implementation details of Oscar[1], an open source implementation of the OSGi framework, to try to provide a deeper understanding of some of the issues at hand, followed by issues that have arisen after much work using and developing for the OSGi framework. Finally, the paper presents the authors' proposed future work to address some of the presented issues, both by improving the Oscar implementation and by providing new functionality on top of the OSGi framework.

II. OSGI SERVICE FRAMEWORK

There have been three versions of the OSGi specification. The core service framework has not changed significantly since the first version; most changes have taken place in the service definitions and these will not be discussed in this paper. The OSGi specification defines the service framework to include a minimal component model, management services for the components, and a service registry. Services (i.e., Java interfaces) are packaged along with their implementations and their associated resources into *bundles*. Services are deployed, as bundles, into the OSGi framework via wide-area networks (i.e., the Internet) or other means (e.g., GSM or memory cards).

The OSGi framework creates a host environment for managing bundles and the services they provide; a bundle is the physical unit of deployment in OSGi and is also a logical concept used by the framework to internally represent the service implementations. Concretely, a bundle is a Java JAR file that contains a manifest and some combination of Java class files, native code, and associated resources. The manifest of the bundle JAR file contains meta-data describing, among other things, the Java packages that the bundle requires or provides.

To be used, a bundle must be installed into the framework. An installed bundle is uniquely identifiable by either its bundle identifier (a number assigned dynamically by the framework when the bundle is installed) or by its location (which is an arbitrary character string used when installing the bundle). The location string is used to retrieve the bundle JAR file and is often an URL from which the bundle can be retrieved. Since bundles are uniquely identified by their location string, it is not possible to install two or more bundles

1 Available at http://oscar-osgi.sourceforge.net.

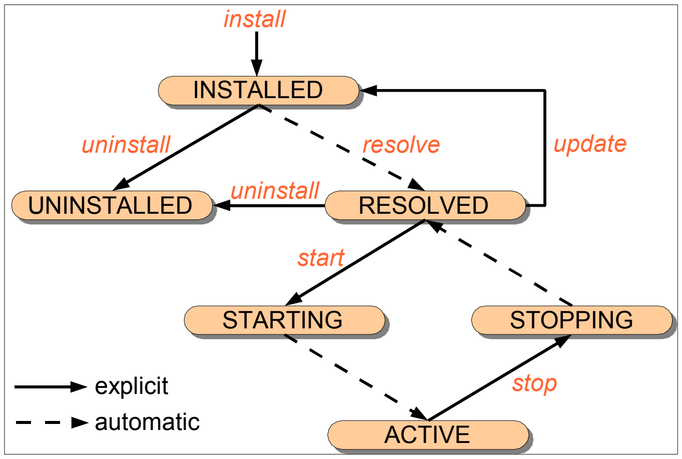

Figure 1. Bundle life cycle states.

from the same location; thus, a bundle is essentially a singleton.

The management mechanisms provided by the framework allow for the installation, activation, deactivation, update, and removal of bundles (see figure 1 for the complete bundle life cycle). When a bundle is installed, it deploys a single component, called an activator, that can register and/or use services. When a bundle is activated, its corresponding activator component is created by the framework. The activator implements activation and deactivation methods that are called to initialize and de-initialize it, respectively. In the activation/deactivation methods, the activator receives a context object, which gives it access to the framework and the service registry. The context allows the activator component to register services, look for other services, and register itself as a listener to different types of events that the framework may fire. When registering a service, the activator component may attach a set of attribute-value pairs to the service. Many different implementations of the same service can be registered by many different activator components and the associated service properties can be used to differentiate among them. To look for a service, an activator component uses the fully qualified service name and an optional selection filter in LDAP query syntax over the service properties.

A bundle may be activated or deactivated at any moment while the framework is running. When a bundle is deactivated, its associated activator component must unregister its services and release the services that it is using. Clients of the activator component's services must take care to observe the departure of the services. Upon departure, clients must release references to the departing services and take any necessary corrective actions. The OSGi framework uses event notifications to signal service arrival and departure.

In order to support bundle deployment and service interaction, the OSGi framework provides sophisticated class loading mechanisms. Bundle meta-data allows bundles to declare imported and exported Java packages that enable sharing of code among installed bundle. This type of code sharing is convenient for libraries, but is also necessary to enable interaction via common service interfaces. The OSGi framework automatically manages the import/export package dependencies through a process called bundle resolution. In OSGi terms, resolving a bundle means matching all of a bundle's imported packages to exported packages provided by other bundles installed in the framework. A bundle can only export classes if it is currently resolved and a bundle can only be activated if it is resolved.

III. OSCAR – AN OSGI IMPLEMENTATION

Oscar is an open source implementation of the OSGi service framework and has been available since the year 2001. Technically, the OSGi service framework can be boiled down to a custom, dynamic Java class loader and a service registry that is globally accessible within a single Java virtual machine. The custom class loader maintains a set of dynamically changing bundles that share classes and resources with each other and interact via services published in the global service registry.

As defined by the OSGi specification, each bundle is represented in memory as an instance of the `Bundle` interface, which provides access to bundle state information as well as bundle operations, such as update and uninstall. Oscar associates a state object with each bundle, which is used to maintain all of a given bundle's run-time state. Oscar's follows a centralized implementation, where one main class, `Oscar`, handles the following tasks:

- Maintains the set of installed bundles,
- Maintains registered event listeners for all OSGi-defined events,
- Resolves imported classes from available exported classes,
- Provides the functionality for all bundle and framework-related operations; specifically, all methods exposed by the two main OSGi interfaces: `Bundle` and `BundleContext`, and
- Provides a bundle cache.

Not all of the above tasks are directly implemented within the `Oscar` class, some tasks are delegated to other classes. Regardless, all functionality must pass through the `Oscar` class at some point. This centralized approach was not always used within Oscar, a decentralized approach and a hybrid approach were both used at various times, but both proved to be too cumbersome. In particular, such approaches complicated event management and concurrency control, since these issues have both local and global implications. Further, complex operations that span multiple bundles, like refreshing a set of bundles after update and/or uninstall operations, were difficult to coordinate. The centralized approach was adopted to overcome these issues. The remainder of this section discusses Oscar class loading in more detail, since it is the most important and complicated aspect of the OSGi framework.

A. Class Loading

Class loading [7] is central to the OSGi framework since it is necessary to activate bundles and to enable interaction among bundles via common service interface definitions. Originally, class loading in Oscar was handled by a custom class loader that was specifically tailored to OSGi. Current versions of Oscar use a generic, policy-driven class loader, called the Module Loader. The Module Loader was created within the Oscar project, but was not specifically targeted for OSGi. The main goal is to provide a generic and extensible class loader that is useful for the many types of projects, such as plugin systems, application servers, and component frameworks, that require more sophisticated class loaders than the standard Java class loaders.

The Module Loader tries to limit the number of class loading policy assumptions it makes, but at a minimum it assumes that sources for classes, resources, and native libraries are grouped into *modules*; modules are only a logical grouping and do not prescribe a packaging or deployment unit format. A *module manager* is used to create and access modules. Each module in a given module manager has an associated *module class loader*, which is used to load classes and resources from the module. To use the Module Loader, an application creates an instance of the module manager, populates it with modules, retrieves the class loader for a particular module, and uses this class loader to load and instantiate application classes. The actual mechanics of how a class is loaded are not precisely defined by the Module Loader.

When the module class loader receives a request to load a class, it does not immediately search its associated module's sources. Instead, it delegates the request to the module manager's *search policy*, similar to how standard Java class loaders delegate class loading requests to their parent class loader. The search policy defines the actual semantics of how a class or resource is found in response to a specific request. The goal is to create a standard library of search policies, which are akin to *class loading patterns*, that are useful to many different projects that require custom class loaders. Some examples of common class loading patterns are *self-contained*, *exhaustive*, and *import/export*. A self-contained policy only looks at the sources of the module associated with the instigating module class loader, as is the case with applets in a web browser. An exhaustive policy looks sequentially into all modules of the associated module manager, similarly to how the typical Java `CLASSPATH` approach works. The import/export policy is a more complex example that enables sharing among modules based on import/export rules.

The import/export search policy allows modules to define required imports and provided exports. It then automatically attempts to validate modules by matching imports to exports. Classes can only be loaded from modules that are valid (i.e., their imports are satisfied). The import/export search policy is parameterized by two additional policies: *compatibility* and *selection* policies. The compatibility policy is used to determine when imports/exports are compatible; this allows the application to use its own definition of backwards compatibility, for example. The selection policy allows the application to select the precise module used to resolve an import; this allows the application to apply scoping rules, for example.

OSGi and the Module Loader

Oscar uses the Module Loader for all of its class, resource, and native library loading. Oscar maps the OSGi concept of a bundle directly to the module concept of the Module Loader. It uses the import/export search policy, since this policy most closely resembles the OSGi class loading pattern. In using the import/export search policy, Oscar provides OSGi-specific compatibility and selection policies. The compatibility policy assumes that imports/exports are versioned Java packages and that all packages are backwards compatible. The selection policy enforces that all modules that import a specific package, get that package from the same export module. To support dynamic package imports, as defined in the OSGi 3 specification, Oscar subclasses the import/export search policy and adds functionality to dynamically import packages when appropriate, if the base import/export search policy fails to resolve a request.

Using the Module Loader in this fashion gives benefits to the Oscar implementation, namely a clearer separation between policy and mechanism. This results in source code that is more understandable, since the specific policy implementations are explicitly separated from other functionality. It also simplifies experimenting with other policies.

Resolving Bundles

A bundle must be resolved before it can be activated; resolving is the process of matching imported packages to available exported packages provided by other installed bundles. The OSGi specification gives some leeway to the framework implementation in determining precisely when to resolve a bundle, as long as it happens before the bundle is activated. Oscar uses a lazy approach to resolving bundles; Oscar only tries to resolve a bundle in two situations:

1. The bundle itself is being activated or
2. The bundle is exporting a package that is needed by another bundle that is being activated.

These two situations are clearly related. If a bundle is activated, it is likely that the transitive closure of packages dependencies will also need to be resolved. The benefit of this lazy approach is that it eliminates start-up ordering issues, which can also ensure that the newest version of an exported package is used if multiple bundles provide different versions of the same package.

The underlying import/export search policy of the Module Loader supports lazy resolution of modules; this process is called validation at the import/export search policy level. To manually resolve a bundle, Oscar accesses the import/export search policy to instigate the validation process on a specific module that has a one-to-one mapping with a bundle at the OSGi level. The import/export search policy also automatically validates modules when an attempt is made to load classes from a module class loader. In this case, Oscar must listen for validation events generated by the import/export search policy so that it can update the state of the corresponding bundle.

IV. OSGi Issues

By using the OSGi framework as a basis for research work [3][5], certain issues were uncovered and/or their importance was magnified. This section is not implying that any of the issues contained herein are flaws of the OSGi framework; rather, it is highlighting issues that are not necessarily readily apparent from reading the specification and that could have an impact on an application that uses or is built on top of the OSGi framework. The issues are presented here in no particular order.

Simplistic package compatibility semantics. OSGi enables sharing of Java packages among bundles using an import/export approach, but it adopts a simple package compatibility semantics that requires that all newer versions of a package be backwards compatible with older package versions. Such an approach does not always reflect real-world constraints, but it is not possible to overcome this limitation except by renaming incompatible package versions (but this is only a partial solution to the problem and requires ownership of the packages in order to rename them). Other projects and technologies, such as Eclipse [6] and .NET [13], explicitly allow for the possibility of shared code not being backwards compatible.

Inflexible package sharing. OSGi only provides for two levels of package visibility, either global or private. Global visibility refers to exported packages that are shared; in this case, all packages that import an exported package must import the same version supplied by the same bundle. It is not possible to support multiple versions of shared packages in memory at the same time. Using private visibility, however, it is possible to have multiple versions of a package in memory at the same time, but, by definition, private packages cannot be shared. For OSGi, this issue is mitigated since backwards compatibility among packages is assumed, but again, this is a real need as witnessed by the fact that both Eclipse and .NET specifically allow for this possibility. This issue is tied to the last issue of package compatibility. If more flexible package compatibility semantics are required, then it is likely that more flexible package sharing approaches will also be required. By providing more flexibility in this area, it would be possible, for example, to upgrade an application partition to a new version of a package, while leaving other partitions of the application using the older version.

This issue of semi-rigid package sharing impacts more than just the ability to have multiple packages in memory at the same time. Take for example, the concept of plugin fragments in Eclipse. This concept allows a plugin to host "plugin fragments," which are essentially merged with the hosting plugin and have access to the hosting plugin's classes as well as any other fragments classes. These types of sophisticated sharing scenarios are not easily supported in OSGi, since all sharing is accomplished in the global scope and access cannot be limited to a sub-scope.

Manual service dependency resolution. The OSGi specification clearly defines how the OSGi framework must automate the resolution and management of imported and exported Java packages among bundles. This same rigor is, however, not applied to the management and resolution of service dependencies. Since bundles interact through services provided by other bundles, it is very likely that any given bundle will have a reasonably complex set of service dependencies. This means that bundle developers must write complex and error-prone code to manage service dependencies for each bundle created.

In the OSGi 2 specification, the OSGi organization introduced the `ServiceTracker` utility, which is intended to simplify the management of service dependencies. Unfortunately, it is cumbersome to use and still requires manual effort on the part of the bundle developer. A technology that demonstrates the feasibility of more fully automated service dependencies is available in the form of the Service Binder [4]. The Service Binder allows bundle developers to describe a bundle's service dependencies in a declarative XML file and then automatically manages those dependencies and the bundle instance at run time. At this point, the OSGi organization has not considered any similar comprehensive support for service dependency automation, although it has introduced a service for "wiring" services that is described in section VI.

Manual resource discovery. The OSGi specification defines how the framework automatically resolves imported packages to locally deployed exported packages, but it does not define a standard way to discover and deploy required packages from external sources. As a result, it is the responsibility of the deployer to manually traverse the transitive closure of package dependencies, find bundles that provide the required packages, and deploy them. This can be realized through an OSGi management console provided by a third party, but these types of tools are proprietary and promote vendor lock-in. The lack of standard resource discovery mechanisms also applies to service dependencies; even if service dependencies are automated, using technology like the Service Binder described above, this only helps if the services are already deployed locally. There is no standard way to discover which services are available for deployment.

Flat service registry. The service registry of the OSGi framework is a simple, flat collection of service implementations offered by installed bundles. As a result, all services offered by any given bundle are visible to other installed bundles. A form of service visibility control can be achieved via security permissions, but this approach is somewhat coarse grained, since it enforces visibility on the service interface name level. This means that, for a specific bundle, service visibility is a boolean flag: either the bundle can see all instances of a given service interface or it can see none. The possibility to limit a bundle from seeing some instances of a given service interface, while still being able to see others of the same service interface does not exist. The need for a hierarchical or scoped service registry is not necessarily one of security concerns, but one of predictable or controlled service composition. This is also related to the next issue.

No service composition level. An application on top of the OSGi framework is assembled dynamically from a collection of bundles interacting with each other via provided services. This model is convenient because application building blocks are loosely coupled, but this loose coupling can have drawbacks. Since the OSGi framework allows bundles to register any number of implementations for a given service interface, it is possible for multiple candidate service implementations to exist at the moment a service dependency is being resolved. This leads to ambiguity in knowing which service to select, in particular when selection filters are not precise enough. The OSGi specification defines an algorithm for selecting a service when multiple candidates are available that is based on service rankings assigned by the bundles themselves or on the order in which services were registered, but these approaches are not sufficient to express preferences for a particular service implementation if it is available, for example.

Even if an application can accept the default OSGi service selection policy, for reasons of predictability, it is likely that the application will have to manually manage and record which service implementation it was using (via the service PID, which persistently identifies a service instance) in order to ensure it keeps using the same service after service changes occur in the framework. The reason the application cannot guarantee predictability by default is because it cannot control whether a service with a higher ranking will be introduced at a later time. To better understand this scenario, consider a service that provides a servlet user interface to manage some device. If an application wants to use such a service, it would most likely prefer to continue to use the same service instance that it was bound to when it was initially invoked, instead of switching at each invocation to the highest ranked service. Switching to the highest ranked service might confuse the end user, if he or she is presented with a slightly different user interface each time the application is used.

Complicated concurrency/class loading issues. Sun Microsystems' Java virtual machine (version 1.4.2) was implemented under the assumption that class loaders have a hierarchical structure, where each class loader can have a parent class loader all the way up to the root or system class loader. During implicit class loading, where a thread is causing classes to be loaded as it executes code, the virtual machine acquires a lock on the class loader associated with the code that the thread is executing. In a hierarchical class loader structure, this is acceptable since the highest thread in the tree will always be able to complete, thus allowing other lower threads to eventually complete.

The class loading structure defined in the OSGi specification, however, is a graph that may contain cycles. Because of this, it is not possible to avoid deadlock in all scenarios, since class loaders may make "horizontal" class loading requests to each other. If two threads are executing code from two different bundles that have a cyclical package dependency, it is possible for them to deadlock when trying to load classes from each other because the virtual machine implementation will acquire the lock on the class loader for each thread and then try to acquire the lock on the opposite class loader when the "horizontal" class loading request is made. It is unclear if this issue affects all Java virtual machine implementations, but Sun Microsystems is aware of this issue in their virtual machine[2].

V. Future Work

It is unclear whether any of the issues described in the previous section will ever be addressed or if they should be addressed by the OSGi specification. Regardless, it is important that the issues are known and understood. Further, it is possible that OSGi framework implementations can be amenable to experimentation in these areas. For example, because Oscar uses the Module Loader, which clearly separates out the OSGi policy decisions, it is easy to modify, tweak, and experiment with alternative policies.

The Module Loader is only the first step towards breaking down a service-oriented platform into its core pieces. Besides class loading, there are at least two other core pieces to a service platform: a deployment mechanism and a service registry. The goal for future Oscar implementations is to try to define these service platform pieces in generic, policy-driven terms, similar to the Module Loader. Upon completion of such a goal, then the OSGi service framework would simply become a (hopefully) thin personality layer on top of these core service platform pieces.

In addition to this goal, an extension of the Service Binder work is underway to provide more sophisticated service composition mechanisms to facilitate complex, but predictable application composition. Lastly, a student thesis to define a resource discovery service for discovering packages and services for OSGi bundles was recently completed, but some work still needs to be done to smooth the rough edges [12].

VI. Related Work

Commercial implementations of the OSGi framework exist and recently another open source implementation was released [9]; a full comparison of Oscar's implementation to these OSGi framework implementations is outside the scope of this paper. Other projects and technologies also target the same area as OSGi, i.e., component and service platforms designed for use in restricted environments; some of these include PECOS, Robocop and Jini and are discussed below. The OSGi Wiring service, discussed below, is related to the issue of service composition.

The goal of PECOS [8] is to enable component-based software development of embedded systems by providing an environment that supports the specification, composition, configuration checking, and deployment of embedded systems built from software components. PECOS defines a component model that supports hierarchical compositions, although these compositions are exclusively static. The goal of the PECOS project is to ensure that component compositions are correct and for this purpose, the notion of rules (i.e., statically checkable constraints) is introduced as a means to provide stronger correctness checks than mere syntactical checks or type checking. Pecos applications however exhibit no dynamic behavior.

Robocop [1] is a project whose aim is to define an open, component-based partial architecture for the middleware layer in high-volume embedded appliances that enables robust and reliable operation, upgrading and extension, and component trading. The appliances targeted by Robocop are consumer terminals such as mobile phones, set-top boxes, and network gateways. Robocop components, which are similar to Microsoft COM components, are delivered as a set of models that include the executable code, simulation models, and behavior models. Robocop's execution environment supports run-time component replacement and integration.

Jini [2] provides an environment for creating applications out of distributed services provided, for example, by physical devices. The service concept of Jini is extremely similar to that of OSGi, since Jini services are also described as Java interfaces. The differences are that services in Jini are organized into named groups and that Jini is a distributed infrastructure which supports the existence of multiple registries. When a service requester finds a service it wishes to use, it receives a proxy that is responsible for communicating with the actual service provider through mechanisms such as RMI. Jini introduces the concept of leasing as a mechanism

2 A bug report concerning this issue was posted in April 2002 here: http://developer.java.sun.com/developer/bugParade/bugs/4670071.html.

used to give access to resources over a period of time in an agreed upon manner. Jini does, however, not define the way services should be composed or the way providers and requesters should be deployed.

The OSGi [10] Wiring Service is a service that provides a mechanism to compose OSGi services. Services are categorized either as producers or consumers and are connected between each other through connectors called *wires*. A wire is specifically created for a particular producer and consumer and the association between these two services occurs at the moment that the services become available. Wires can be created and bound to producers and consumers during run time, but there is no way to describe an application as a topology.

VII. Conclusion

The OSGi service framework provides a simple, lightweight framework for creating service-oriented applications. Because of these characteristics, the OSGi framework is seeing an increased interest in applying it to both consumer device and non-consumer device domain areas. This paper presented details of a specific implementation of the OSGi framework, called Oscar, in order to shed some light on the framework's internal workings. The authors' experience implementing the OSGi framework lead to a deeper understanding of more general service platform issues that were also presented in this paper. The intent of this paper was not to evaluate the OSGi framework directly, but to open a general service platform discussion. The OSGi framework and other related service platforms show great potential for advancing how applications of the future are created.

References

[1] C. Aarts, "*Robocop: Robust Open Component Based Software Architecture for Configurable Devices Project*," Framework Concepts, Public Document, May 2002.

[2] K. Arnold et al., "*The Jini Specification*," Addison-Wesley, 1999.

[3] H. Cervantes and R. S. Hall, "*Beanome: A Component Model for the OSGi Framework*," Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices, September 2000.

[4] H. Cervantes and R.S. Hall, "*Automating Service Dependency Management in a Service-Oriented Component Model*," In the proceedings of CBSE6 Workshop, May 2003.

[5] R. S. Hall and H. Cervantes, "*Gravity: Supporting Dynamically Available Services in Client-Side Applications*," Poster paper in Proceedings of ESEC/FSE 2003, September 2003.

[6] IBM Corp., "*Eclipse Platform Plug-in Developer Guide*," Online Whitepaper, 2000.

[7] S. Liang and G. Bracha, "*Dynamic class loading in the Java virtual machine*," Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98), 1998.

[8] P. Mueller, C. Zeidler, C. Stich, and A. Stelter, "*PECOS - Pervasive Component Systems*," Workshop on Open Source Technologie in der Automatisierungstechnik, 2001.

[9] Opensugar, "*JEFFREE – Java Embedded Framework Free*," http://forge.objectweb.org/projects/jeffree, 2003.

[10] Open Services Gateway Initiative, "*OSGi Service Platform Specification*," Version 3, March 2003.

[11] Open Services Gateway Initiative, "*Official web site*," http://www.osgi.org, 2003.

[12] K. Pauls, "*Eureka – An OSGi Resource Discovery Service*," Thesis, Free University Berlin, September 2003.

[13] D.S. Platt, "*Introducing Microsoft .NET*," Microsoft Press, Second Edition, 2002.