A Repository for Integration of Software Artifacts with Dependency Resolution and Federation Support

Rodrigo García-Carmona, Félix Cuadrado, Juan C. Dueñas, and Álvaro Navas

Departamento de Ingeniería de Sistemas Telemáticos, ETSI Telecomunicación,
Universidad Politécnica de Madrid, Madrid, Spain
{rodrigo,fcuadrado,jcduenas,anavas}@dit.upm.es

Abstract. While developing new IT products, reusability of existing components is a key aspect that can considerably improve the success rate. This fact has become even more important with the rise of the open source paradigm. However, integrating different products and technologies is not always an easy task. Different communities employ different standards and tools, and most times is not clear which dependencies a particular piece of software has. This is exacerbated by the transitive nature of these dependencies, making component integration a complicated affair. To help reducing this complexity we propose a model-based repository, capable of automatically resolve the required dependencies. This repository needs to be expandable, so new constraints can be analyzed, and also have federation support, for the integration with other sources of artifacts. The solution we propose achieves these working with OSGi components and using OSGi itself.

Keywords: Component Distribution, Model-driven Engineering, OSGi, Open Source, Software Integration.

1 Introduction

In recent years software development has been undergoing a huge change, evolving from a closed software paradigm to new processes that incorporate open source software in products and services. The number and relevance of software developments based in the open source paradigm have experienced an exponential growth [1].

The reason for this lies in the particular strengths that open source can bring into the table, like its ability to reduce IT costs, deliver products faster and improve the security and reliability of systems. This situation has been also fostered by the numerous success cases in industry that have followed this model. In fact, sources like Forrester have described 2009 as “the year IT professionals realized that open source runs their business”, and predict that this trend is going to continue in the following years [2].

Most of the strengths that open source provide are the product of the open communities and the development models that arise from them. At this point it has become clear that the community efforts are leveraged by the participating elements, with everyone benefitting from the created ecosystem. Example succesful communities are the Apache Software Foundation, the Eclipse Foundation, the ObjectWeb community and SourceForge.

Those communities have matured with different collaboration and architecture models. As a consequence, these communities are like isolated islands which no communication between them. Unfortunately, while they achieve a very high internal consistency, there is a severe lack of compatibility and integration among them. This hampers one of the most important factors for the success of open source; the reusability of code, since the lack of integration complicates this process. These integration challenges are also aggravated by the multiplicity of tools used by different projects. To further complicate this issue, most of these tools are not concerned in working with other solutions.

The heart of this problem lies in evaluating the interdependencies of software components. These dependencies tend to form a complex mesh that can span several projects and code bases and it is difficult and costly to navigate. One of the most severe problems of open source development is figuring which already existing components one has to use.

In addition to the technology impedance mismatch, there are additional factors which must be considered. An often overlooked factor with open source usage is the existence of several software licenses. While on first thought these elements should not interfere, they do actually restrict the potential uses, since some of these licenses are incompatible between them or with commercial ones.

It can be seen that the problem lies not only in code interoperability but also in additional aspects, such as legal license compatibility, or design according to similar hardware capabilities. In practice, all these problems tend to produce fragmentation, complicating the use of what software it is already available.

The meeting point for this integration is, in almost every community, a software repository. Since the repository act as a central hub for all development efforts, the difficulties exposed here are particularly evident in it.

In this article we present a comprehensive, model-based component repository that provides two features that ease the integration of software elements: 1) An automatic dependency resolution that can work with several types of concerns (software, hardware, etc...) and 2) A federation system that can aggregate the contents of other repositories and in turn expose their own components to the outside world.

This repository has been developed in the context of the ITEA-OSAMI European project and its objective is to act as a main hub in an Internet federation of repositories, while being publicly available. It integrates artifacts published by the members of both the ITEA-OSAMI project itself and external partners.

This article is structured as follows: The next section gives a brief explanation of the most recent developments in the topics that concern our work. Section 3 explains our proposal in detail, and section 4 performs a validation of our work using an implementation of the repository. Finally, the last section outlines the conclusions we have reached and shows how our work could be further developed in the future.

2 State of the Art

In this section we provide a brief state of the art of the technologies that are especially relevant for our proposal: the OSGi component model and the already existing software repository solutions.

2.1 The OSGi Component and Service Model

OSGi is an open specification that defines a component and service model for the Java platform. The latest version of the specification is 4.2 [3], and is maintained by the OSGi Alliance, a consortium formed by embedded and enterprise companies, such as IBM, Oracle, Red Hat or Siemens. It was originally designed for home gateways and embedded systems, but its adoption has greatly increased lately in desktop tools and enterprise application servers.

The relevance of the OSGI specification has increased mainly because it provides a modularity layer that was missing in the Java platform. This is enabled by the definition of OSGi bundles. Bundles extend java libraries (JAR files), allowing them to expose their functionality to the rest of the platform in a controlled way. Bundles use the Java manifest file to declare explicitly what does the component provide to the rest of elements (in the form of java packages) and what does it require in order to work properly (either java packages or complete bundles), providing in both cases version compatibility information. This directly addresses the 'JAR hell' problem of complex Java-based systems, greatly easing the deployment and configuration of new software.

Additionally, OSGi bundles collaborate through a lightweight service mechanism, with services being runtime Java objects that implement interfaces. This enables effective decoupling between collaborating components and simplifies the development of extensible systems. The OSGi framework provides an execution platform for OSGi bundles, enabling dynamic deployment and configuration of the components. These factors make OSGi bundles an ideal specification for open, composable service-based ecosystems, as it provides simple mechanisms for effective interoperability and modularization.

All in all, OSGi is the best solution in the Java world for the design and implementation of modular applications. It enables an even lower coupling and brings the SOA principles to the virtual machine. Our proposal will use OSGi for both the components the repository will manage and the actual repository itself.

2.2 Software Repository Standards

At this moment there are several repository technologies that are popular in open source communities. Every one of them has its own component model and capabilities. We detail each in this section, with a special focus in their support of OSGi bundles and federation capabilities.

Maven [4] is one of these solutions, a software project management and comprehension tool. Maven has become the de facto standard for managing Java projects, thanks mainly to the support and its extended use inside the Apache community. Maven uses a generic project description model for describing software projects named Project Object Model (POM). The POM file of a project defines the project's lifecycle as well as its dependencies and configuration parameters. However, this model has been defined as generic as possible, in order to cover a wide range of software projects. Hence, it does not accurately reflect the special relationships of specific types of software components, such as OSGi bundles. For example, dependencies onto a particular software package can be defined, but not onto a complete bundle.

POM cannot describe these kinds of dependencies, losing information in going from manifest to POM. Despite this disadvantage, Maven repositories provide other interesting capabilities, such as being able to store all the information concerning a project (source code, documentation, etc) or the hierarchical federation with other Maven repositories, augmenting the Maven basic dependency resolution mechanism. However, this mechanism does not work with repositories implemented using other technologies.

Another model used to describe bundles is the OBR (OSGi Bundle Repository) project. This model was presented as the draft OSGi RFC 112 [5]. The RFC defines both an XML schema for bundle description and the Java API for browsing OBR repositories. An OBR repository is very simple in its structure, providing just an XML file describing the server contents. This eases the creation of OBR repositories as only the bundles and how to download them need to be described, leaving plenty of freedom to design the architecture supporting those operations. This simplicity has the drawback that the clients are forced to carry out most of the operations, a problem aggravated by the fact that there is no standard definition of an OBR client. The draft status of the OBR presents additional disadvantages, such as the lack of mechanisms for managing repository contents (e.g. upload new bundle, update, or delete). and that the federation mechanism between OBR repositories is not well-defined.

In the 3.0 version of Eclipse, the Eclipse architecture was changed to use OSGi as the project core. This change pushed the Eclipse community to develop their own bundle repository, named P2 [6]. The P2 repository is widely used, since version 3.4 of the Eclipse Platform uses it as the management mechanism for its components (OSGi bundles). The P2 specification defines two repository types: metadata and artifact. The metadata repository stores Installable Units, which are the P2 representation of an artifact. This means that almost anything can be described as an Installable Unit (configuration files, bundles, executables, etc). The metadata repository also provides the P2 federation mechanism. Complementing it is the artifact repository, which stores the binary and description files associated to the Installable Units. There is also a third component, the Director, which is part of the repository client. The Director is in charge of resolving dependencies and installing and uninstalling the artifacts. However, this solution has an important drawback: Its component model is concerned only with software direct dependencies, being oblivious to other constraints that could affect artifacts.

Also, the increasing success of the OSGi platform has stimulated the creation of proprietary bundle repositories especially dedicated to store this type of software components. The Spring Bundle Repository [7] is the most notorious example of this trend. This repository stores a collection of bundles and library description files ready for production use. A library description file is a document describing a set of bundles that are frequently used together. The access to the repository is made through Maven, Ivy or a web interface. The web interface shows information related to the dependencies and exported resources of a bundle, offering links to download them. However, the proprietary nature of this solution greatly limits its applicability and usefulness.

It can be seen that there are a lot of existing solutions for a bundle repository. But with the exception of the Spring repository (which supports Maven), there are no federation mechanisms between repositories of different types. On top of that, different development communities have chosen different repository solutions. This fact makes implementing a dependency resolution mechanism a difficult task.

Despite a previous attempt at creating standard-complying repositories [8], this work has not been followed up since its publication.

However, in the digital contents world there are many studies [9] and proposals [10-12] on this topic. But none of them have been applied to a software artifact repository. There are huge differences in nature and needs between software components and multimedia contents, and the solution that works with one cannot be used with the other without severe modifications.

3 Proposed Solution

In this section we detail our proposed solution. For this aim we have divided this chapter in several subsections.

As we have already said in the introduction, our aim was to provide an artifact repository that helps to improve software integration. To achieve this, our solution provides two main features: A faceted dependency resolution, and a repository federation engine. One subsection is devoted to each of them.

Also, to fully grasp how we propose to fulfill both, first we introduce two basic topics needed for the proper understanding of our proposed repository: 1) the characteristics of the model representation of software artifacts, and 2) the architecture of the repository itself.

3.1 Software Component Metamodel

To enable the correct processing of the components the repository needs to manage, and the integration of information to and from other solutions, it is imperative to have a model representation of the software elements. Therefore, we have defined a metamodel with enough expressivity to capture all the information that we need, but at the same time hiding non needed data.

From this metamodel, a model instance describing each software artefact, its capabilities and its needs can be created. We have named these model instances Deployment Descriptors.

Although the metamodel will be primarily used to represent OSGi-related artefacts, it has been designed to support without modifications other elements, such as non-bundle JARs or additional component/service models (such as EJBs, Spring beans or Web Services). The metamodel aims to capture all the relevant information of all types of software elements. This is enabled by the concept of Resource, which we have adopted from the OMG Deployment & Configuration [13] standard.

Figure 1 depicts a subset of the metamodel. As can be seen in it, Resources are the main building blocks. A Resource represents any logical or physical manageable system element, and it is defined by three fixed parameters (name, version and type) common to all Resources and an undefined list of specific Properties for each Resource type.

The core element is the Deployment Unit. Deployment Units represent the artifacts which can be deployed over the environment containers. In an OSGi context, Deployment Units represent OSGi bundles. Conceptually, a Deployment Unit would be the lowest abstraction level of our software model, being the unit of software distribution. The Deployment Unit is composed by a set of children Resources, Dependencies and Constraints that provide computable information about the developer,

software license, packaging type, exported packages, logical dependencies and hardware compatibility restrictions.

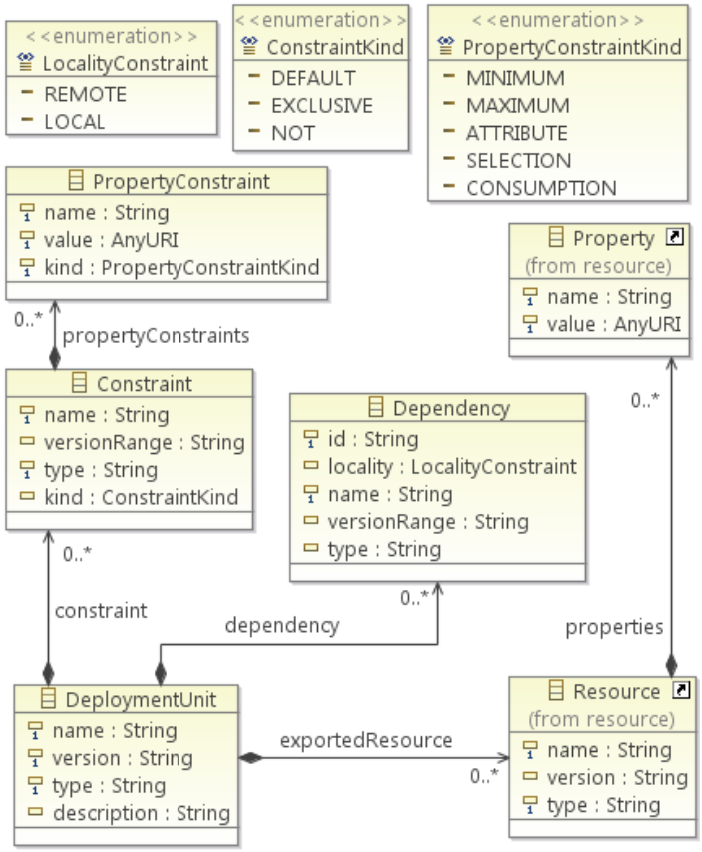

Fig. 1. Software component metamodel

Dependencies represent the required physical and logical dependencies needed in order to assure a smooth running of the Deployment Unit. Two types of elements can satisfy a Dependency: specific Resources or whole Deployment Units. This represents the two main mechanisms defined in OSGi: Require-Bundle and Import-Package. In addition to this, the metamodel allows us to further describe the type of Dependency, enabling to differentiate requirements on remote resources such as Web or REST Services from requirements that must be satisfied by a unit in the same host (e.g. as it provides a Java package). This aspect is identified by a locality parameter that could be remote or local.

Finally, Constraints enable the expression of requirements over the runtime execution environment, each one requiring a specific Resource to be present at (or absent from) the environment. Following this definition, we have defined three kinds of Constraints, depending on the required behaviour: *default* (to be present), *exclusive* (to be present and not used more than once) and *not* (to not be present). To further

extend the Dependency and Constraint models, Properties can be defined. Each needed Property can be defined by a name, an evaluation function and a threshold value.

As an example, a typical Constraint would identify a Resource of type "hardware.processor" with an additional Property "speed" of a kind "minimum" and an expression value of "2000". This means that the Deployment Unit requires a microprocessor with a minimum speed of 2 GHz.

Both Dependencies and Constraints are shown in Figure 2.

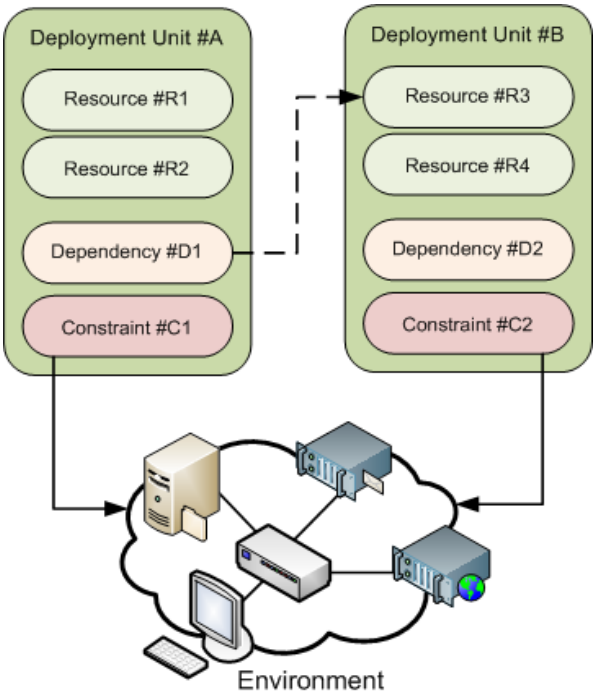

Fig. 2. Dependencies and constraints

The metamodel allows us to represent all the OSGi-specific mechanisms, as well as expressing important information that is not contemplated on the original format (the manifest file) or the information models of other repository solutions. This means that a conversion from one of those solutions into our proposal would not result in the loss of information, although the opposite would.

3.2 Repository Architecture

The need to federate with multiple types of repositories, as well as evaluating component dependency taking into account multiple factors have motivated us to design the architecture of the repository with a modular and extensible approach. We have selected the OSGi platform as the base technology to achieve those requirements. On a side note, this allowed us to test the system from the start, in order to check whether

the repository was able to host itself successfully. Figure 3 shows a layered view of the repository components.

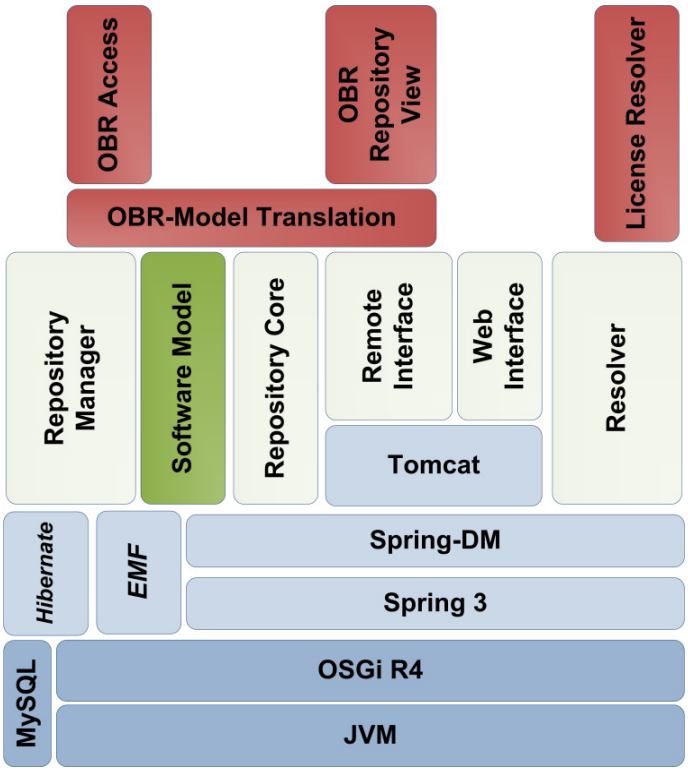

Fig. 3. Repository architecture

In the lower levels lie the Java Virtual Machine, the OSGi framework, and a database solution for storing the relevant information. On top of it are the basic OSGi components from third parties that are needed for the repository to work. The repository uses Spring Dynamic Modules for structuring the inter-bundle communications. The EMF (Eclipse modeling Framework) components enable the definition of the metamodel and provide the tools needed to work with them programmatically. Hibernate Provides an ORM (Object-Relational Mapping) interface with the database system. Finally, the Apache Tomcat bundles embed a lightweight application server that will host the remote access interfaces developed for the repository.

The next layer contains the basic components of the repository:

- Software Model: Provides the metamodel defined at the previous section, as well as Java bindings and marshalling mechanisms.
- Repository Core: The basic component of the repository. Provides the main service interfaces of the components and defines the extension semantics.

- Repository Manager: This component manages the physical artifacts and the component information. It provides CRUD (Create-Read-Update-Delete) operations over the managed Deployment Units,
- Web Interface: Web-based graphical user interface that allows human users to browse the repository contents.
- Remote Interface: Exposes a REST interface that enables the communication between the repository and other software.
- Resolver: The component that processes and resolves Deployment Unit dependencies, obtaining unit closures that work correctly together.

Finally, the topmost level of the diagram shows some extensions to the repository that expand the base functionality to federate with an additional type of repositories (OBR), and apply additional criteria for the dependency resolution. Over the next sections we will present additional details on the federation and resolution capabilities of the repository.

3.3 Faceted Dependency Resolution

This modular architecture makes possible the easy expansion of the repository capabilities. This feature is used to define different types of dependencies, each one resolved by a different component. Moreover, this structure enables the definition of a faceted dependency resolution engine. In it, there are not only several dependency types, but also additional conditions that the candidates to satisfy one need to comply. These conditions are called Facets.

An example of a Facet is the license compatibility. It is perfectly possible that a Deployment Unit satisfies every dependency that another has, but at the same time do not be valid because their licenses are incompatible. Other Facets could be security settings, packaging procedures or execution requirements. Each Facet can be added to the resolution engine as an OSGi bundle, and it offers its features as services that are called by the resolution core component.

An example of this process that uses a License Compatibility Facet we developed is shown in Figure 4. In it a user calls the resolver with the intention of knowing the dependencies needed for a Deployment Unit (DU). The resolver processes every Dependency, looking for other Deployment Units that can satisfy it. After some of them have been found, the resolver searches for Facets that need to be checked and, after finding one (License Compatibility), makes use of it. In this particular example only one unit is valid after this check.

The integration of new facets to the dependency resolution is straightforward, and the license compatibility is just one example of application. As the diagram shows, the second loop will check each DU for every detected facet.

3.4 Repository Federation

To enable the integration between open source components it is not enough to just re solve their dependencies. It is also necessary to be able to provide the artifacts that fulfill those requirements.

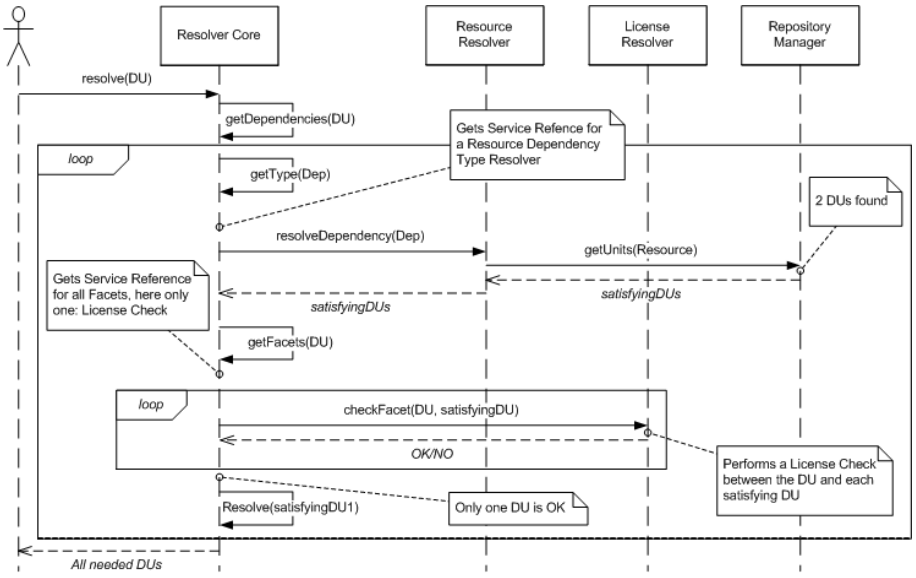

Fig. 4. Faceted dependency resolution

Since open source communities are fragmented and each one uses different tools and techniques, the repository needs to access and understand the information that lies in other repositories. To achieve this end, the federation capabilities allow our repository to communicate with external types of repositories currently available. Federation support is designed with extensibility in mind, and each technology extension can be federated just by providing three services:

- Model transformation service between our information model instances and the format the target solution uses to represent software artifacts.
- Remote manager service that accesses the information contained in the external federated repositories.
- Remote interface service that can be accessed by the target solution repositories. This is only possible if the target solution has some kind of federation support for repositories of its own type.

For a more detailed explanation of how these features can be implemented we will use OBR as an example of a target solution. An example of an infrastructure that uses this two-way federation is depicted in Figure 5.

Sample Integration: OBR. The OSGi Bundle Repository RFC is a draft standard for providing a common interface to distributed OSGi repositories. Its official nature, alignment to OSGi concepts and the explicit acknowledgement of federation requirements make it an ideal candidate for testing our federation approach. Here we present how we achieve two-way integration between our repository instances and federated OBR repositories.

We talk about two-way federation, as we both act as OBR providers and consumers. For external OBR repositories, we offer an OBR view that can be used by them in their standard federated dependency resolution processes. Additionally, our repository can handle a list of external OBR repositories, and can delegate dependency

resolution requests to the distributed OBR instances. Both approaches of federation are achieved though the same method: Model transformation from our generic model to the specific component model defined by OBR.

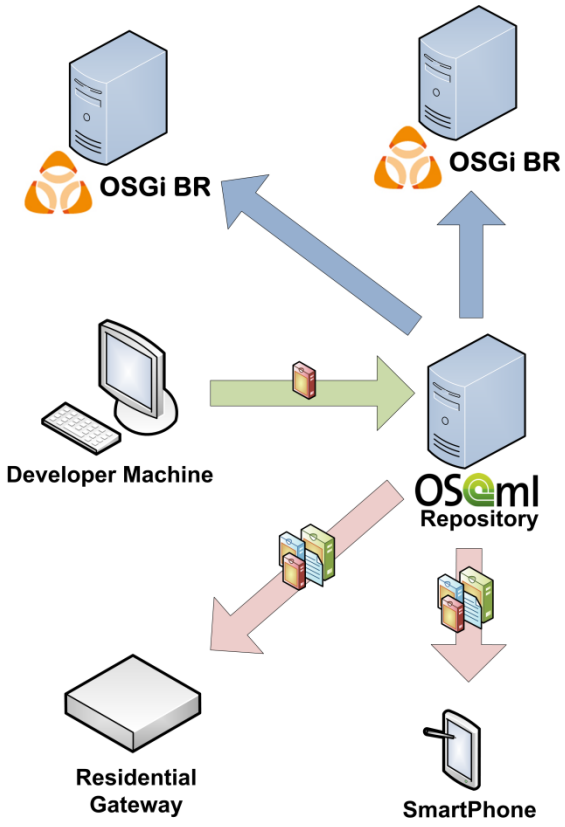

Fig. 5. Repository federation

For maximizing extensibility, the OBR model does not explicitly rely on OSGi concepts. The repository works with resources, identified by a symbolic name and a version. Additionally, a download URL is provided for each element. Each resource contains two types of declarations. First, resources offer a list of capabilities to the environment. They represent either the whole element (named *bundle*), or a software element such as a java package (named *package*). Each capability is further refined through properties, which have a name, value and value type (e.g. String or number). The second kind of elements are requirement statements, that demand the presence of resources in the resolved configuration. They model logical requirements that must be satisfied. This specification's concepts can be mapped to a subset of the Deployment Unit model.

Figure 6 presents an example mapping between both models. Every OBR concept has an equivalent definition in our abstractions. The OBR resource plus the bundle capability elements are mapped to the base Deployment Unit concept (The definition

of units as resource subclasses allows this). Additional capabilities are mapped to unit exported Resources, such as the presented java package. Resource visibility information is lost, which is not problematic for OSGi-specific elements (all of them are local), but presents the limitations of OBR for reasoning over distributed physical environments. Additionally, each OBR requirement is mapped into a logical Dependency. All the information derived from the Constraints from our model has no equivalent. This bears no impact from the OBR perspective, as our repository provides all the required information. On the other hand, the use of federated OBR repositories by our specific instance can result in lesser-quality results, in cases when physical concerns need to be evaluated.

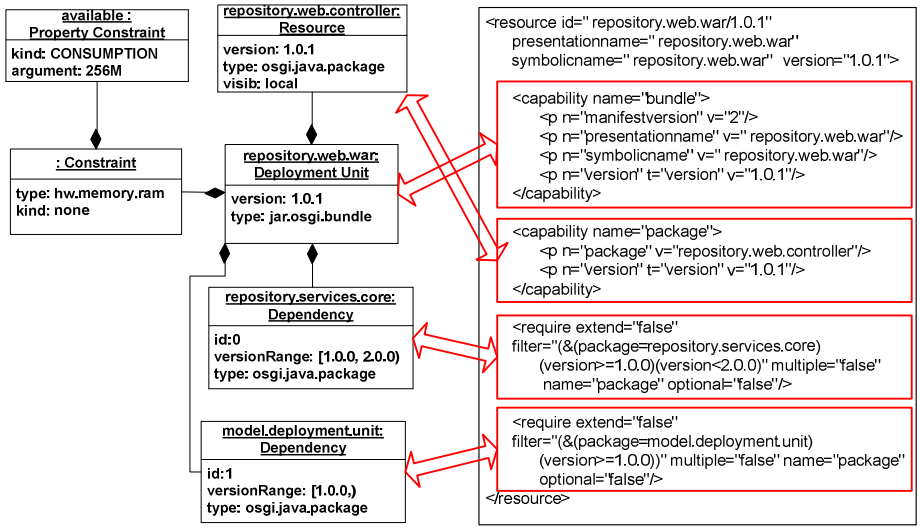

Fig. 6. Model mapping between OBR and the presented model

4 Validation

ITEA-OSAMI is an european project which was executed by 34 partners from both the academia and enterprise. The objective of this project was to develop open source common foundations for a distributed, dynamic service-oriented platform.

Consortium partners came from multiple domains (healthcare, personal, and mobile office), followed different software development processes, and depended on existing open source resources from different communities. This created a need for a centralized platform that eased partner integration.

The repository presented in this article has been developed and deployed in order to address the project requirements. It has widely succeeded at this task, becoming the central point of the ITEA-OSAMI ecosystem. ITEA-OSAMI's running instance of the repository could be publicly accessed at the time of writing of this article[1]. (Figure 7 shows the visual aspect of the web interface, which is listing several units already resolved).

[1] `http://repository.osami-commons.org`

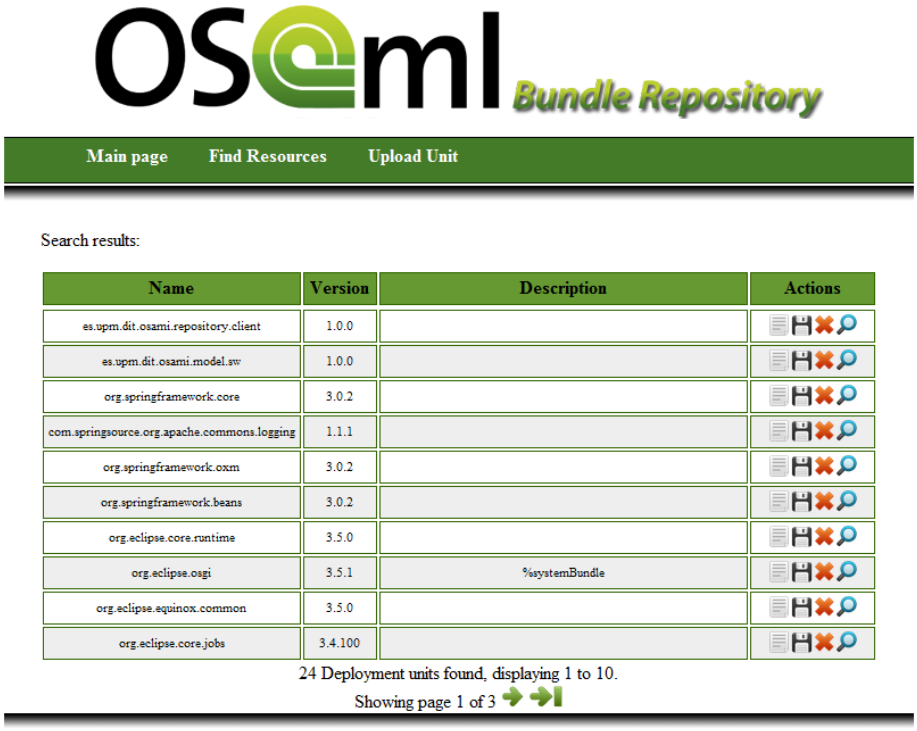

Fig. 7. Screenshot of the repository

After assessing partner concerns, we provided two extensions to the repository. Regarding federation, an OBR extension was developed, as it was mandatory to support OBR-based deployment clients as well as accessing open source bundles developed at two third-party repositories. Additionally, since the beginning of the project open source license management was a potentially conflicting aspect among partners. However, these issues were addressed with the definition of a license-aware dependency Facet, based on the dependency compatibility analysis module from another project partner.

In this context, we have validated the proposed metamodel, as well as the defined architecture and extensibility capabilities. It has successfully been used by all the project partners, providing a common integration point for the developed open source software and services, both internally created and from the main open source communities.

5 Conclusions and Future Work

In this article we have proposed an architecture for a repository for the integration of software artifacts, with a special focus on OSGi bundles. This repository has been designed around an information model for software components, which manages to show all relevant data while hiding the undesired complexities.

We have also shown how this architecture has been created to be extensible. Using this modularity we have demonstrated how it can be easily expanded to support two important features:

- Faceted dependency resolution: Offers support for an unlimited number of conditions that dependencies are forced to respect.
- Two-way federation: Enables our solution to access contents available in other repositories and at the same time expose itself to them.

We also have developed a license compatibility Facet based upon existing open source work and the components needed to achieve federation with OBR repositories.

Finally, our work has been validated in the context of the ITEA-OSAMI European project, where it has been subjected to an intensive use by more than 30 partners from different countries and sources (universities, research centers, software and telecom companies, etc).

Concerning future developments, the most straightforward way to improve the already existing work is through the support of more repository technologies for federation and new dependency Facets. In the first field, federation with Eclipse P2 would be the most interesting repository to support, since it sees wide use in several communities. About dependency Facets, more of them could be developed, taking as an example the license check already created. These components are relatively easy to implement thanks to the infrastructure of our proposal.

Not directly related with this, but also interesting, are the possibilities for the repository to work in a cloud environment. This line of work is concerned not only with the deployment of the repository itself, but also with how it can manage the software artifacts of several types of cloud solutions (IaaS, PaaS or SaaS). To support features like these the information model would probably need to be extended and the architecture of the repository revised.

Using the capabilities of OSGi plus some extensions already in the making[2], the possibility of extending OSGi for a complete PaaS solution is turning into a reality. If this possibility finally materializes, the proposed repository could be expanded to work in this kind of environment.

Acknowledgements. The work presented in this article has been performed in the context of the European project ITEA-OSAMI, under grant by Spanish Ministerio de Industria, Turismo y Comercio in the PROFIT program.

References

1. Deshpande, A., Riehle, D.: The Total Growth of Open Source. In: Russo, B., Damiani, E., Hissam, S., Lundell, B., Succi, G. (eds.) Open Source Systems. IFIP, vol. 275, pp. 197–209. Springer, Boston (2008)
2. Evelson, B., Hammond, J.: The Forrester Wave: Open Source Business Intelligence (BI), Q3 2010, Forrester Research (2010)

[2] `http://www.osgi.org/wiki/uploads/Design/rfp-0133-Cloud_Computing.pdf`

3. OSGi Alliance, OSGi Service Platform Release 4 Version 4.2 Specifications (June 2009)
4. Massol, V., Van Zyl, J., Porter, B., Casey, J., Sanchez, C.: Better builds with Maven. Mergere Inc. (2006)
5. Hall, R.S.: OSGi RFC-0112 Bundle Repository (February 2006)
6. Le Berre, D., Rapicault, P.: Dependency management for the Eclipse ecosystem: Eclipse P2, metadata and resolution. In: Proceedings of the 1st International Workshop on Open Components Ecosystem. ACM (2009)
7. Rubio, D.: Pro Spring Dynamic Modules for OSGi™ Service Platforms. Apress (2009)
8. Iyengar, S.: A universal repository architecture using the OMG UML and MOF. In: Proceedings of the Second International Enterprise Distributed Object Computing Workshop, EDOC 1998 (1998)
9. Kraan, W., Mason, J.: Issues in Federating Repositories, A Report on the First International CORDRAtm Workshop. D-Lib Magazine 11(3) (2005)
10. Smith, M., Barton, M., Bass, M., Branschofsky, M., McClellan, G., Stuve, D., Tansley, R., Walker, J.H.: DSpace, An open Source Dynamic Digital Repository. D-Lib Magazine 9(1) (2003)
11. Van de Sompel, H., Lagoze, C., Bekaert, J., Liu, X., Payette, S., Warner, S.: An Interoperable Fabric for Scholarly Value Chains. D-Lib Magazine 12(10) (2006)
12. Van de Sompel, H., Chute, R., Hoschstenbach, P.: The aDORe federation architecture: digital repositories at scale. International Journal on Digital Libraries 9(2)
13. Object Management Group. Deployment and Configuration of Distributed Component-based Applications Specification. Version 4.0 (April 2006)