

Enhanced OSGi Bundle Updates to Prevent Runtime Exceptions *

Premysl Brada
Department of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic
<brada@kiv.zcu.cz>

Abstract

Explicit declaration of provided and required features facilitates easier updates of components within an application. A necessary precondition is that sufficient and correct meta-data about the component and its features is available. In this paper we describe a method that ensures safe OSGi bundle updates and package bindings despite potentially erroneous meta-data. It uses subtype checks on feature types, implemented as user-space enhancements of the standard bundle update process. The method was successfully applied in the Knopflerfish and Apache Felix frameworks and the paper discusses the general experiences with the OSGi framework gained during the implementation.

1 Introduction

The component paradigm has become a mainstream approach in software engineering. It is generally accepted that components are coarse-grained software building blocks with explicitly declared surface¹ features [14]. The approach is used in many frameworks ranging from pure experimental to industrial ones; among the latter, the OSGi platform [11] is getting increased attention lately.

Components are treated as black boxes and therefore necessarily declare the features they require from outside. The bindings between them can then be established by a component framework (container). This aspect not only leads to a more manageable software decomposition, it also enables to use different implementations interchangeably and facilitates easier updates of application parts. The necessary precondition is that sufficient meta-data about the component and its features is available in the distribution form, since access to the source code is often neither possible nor desirable.

*Preliminary version of an article, to appear in the proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), September 3rd - 5th, 2008.

¹We use this term to prevent ambiguity with *interface* as e.g. in Java.

In this paper we present a method that ensures safe component updates and feature bindings despite potentially incomplete or incorrect meta-data, using analysis of only the current and update component versions. While the problem is a general one, we apply it on the OSGi case (see next subsection) in this paper. Section 2 elaborates on the paper’s goals and provides a motivation example.

The approach we take in addressing the problem entails comparing the types of component features, as an enhancement of the standard component resolution and binding process. The other research works in this area, discussed in Section 3, usually take either a very formal approach or provide simplistic ad-hoc solutions – our aim is a “best of both worlds” position. The details of our approach are described in Section 4 including notes on implementation in two OSGi framework instances.

One of the major obstacles we had to overcome along the way is the lack of standardized access to distribution packages of bundles currently installed in the framework. These and other experiences with the OSGi framework we gained during the implementation are summarized in Section 5.

1.1 Overview of OSGi

The Open Services Gateway Initiative (OSGi) platform [11] is an open Java-based framework for service deployment and management. Its original target were embedded applications (smart appliances, mobile devices), for several years now however it has seen a growing success in large-scale systems.

The core of OSGi Service Platform is the *framework* (called “container” in many other component platforms) which creates a runtime environment for managing the deployment and lifecycle of components called *bundles*. It also provides a registry of services exported by bundles, to enable their registration and lookup, and a security system to authorize access to resources. A set of standardized services [10], implemented by system bundles, is an integral part of the framework specification.

The component model of OSGi is fairly simple. A bundle can declare a number of surface features: Java packages, services accessible through Java interfaces, native libraries used, dependencies on the execution platform and dependencies on concrete bundles. The first two, i.e. packages and services, can be exported (provided) or imported (required). The physical form of a bundle is a JAR archive with a manifest file containing meta-data about the bundle and (some of) its features.

Services, a key OSGi feature, allow dynamic registration, lookup and (un)binding of functionality mediated by a centralised framework registry. Service management is done either manually by bundle code or via several standardized services, the Declarative Services specification in particular. On the other hand, the framework completely manages the binding of packages and the remaining features. The process of finding and attaching an appropriate exported package to an package import declaration is called *resolving* a bundle; it involves a sophisticated use of Java classloaders. Versioned dependencies can be used in resolving, since the framework honors version numbers and ranges attached to package and bundle declarations to express compatibility. No strict rules for interpreting the version numbers are enforced however, the specification only suggests to follow the standard scheme mentioned in Section 3 below.

Bundle lifecycle is well defined and can be easily controlled by both the framework and authorized bundles. The following states are distinguished, among others: *installed*, *resolved* (package dependencies have been satisfied) and *active* (bundle instance has been started, initialized and is ready to provide services); stopping an active bundle makes it transfer to the resolved state.

Bundles can be updated, even at runtime: the process essentially follows a stop-reinstall-start cycle. If the update version cannot be resolved, it is left in the installed state to prevent runtime problems. The framework uses event handling mechanism to announce bundle state transitions and service creation and removal.

2 Motivation

Let us describe the issue we are addressing by type-safe bundle updates. As most component frameworks, OSGi in principle uses straightforward name-based binding of component features, where a required feature is matched to a provided one when their type names (e.g. a fully qualified class name) are equal.

Matching features during component update is a special case of the general binding process. It is done to ensure especially that the clients of the component being updated will be able to work correctly after the update.

The first type-related problem found in updates is when the new component is not a contravariant version of the current one because some exported features are missing (potential problem for component's clients) or new imported ones have been added (problem for the component itself). This issue is usually relatively easy to deal with, and the OSGi bundle resolving process provided by OSGi core together with the declarative services extension handle this case reasonably well².

2.1 The problem addressed

On the next level, update can lead to problems if different (incompatible) versions of a given feature are (re)bound in the resolved import-export pair. Here the versioned dependencies used by OSGi add an extra level of security.

The component provider can specify feature (especially package) versions and version ranges to ensure that only compatible feature types are bound. If the imported feature in the new version cannot be successfully matched against the available set of exporters under the specified version constraints, the bundle is prevented from starting. This is more desirable than a run-time error resulting from a disregarded version mismatch.

The problem with the approach is its fragility. Feature compatibility is determined by matching only feature type names and versions or version ranges. The contents of the types, or rather the correspondence of the (change in the) type contents and the (change in the) version identifiers between versions, is not verified. The ultimate effect is a run-time error when there is a mismatch in type – most often method signature – expected by the importer and provided by the updated exporter.

²If only core framework capabilities are used, the bundle code needs to handle service lookup and dependencies manually with the help of standard service registry.

2.2 Example: OSGi package versions

Consider a simple OSGi component implementing a weather station, connected to several sensors e.g. thermometers. Suppose we have the thermometers accessible as an OSGi bundle with an exported service. Its interface together with corresponding meta-data is shown in Listings 1 and 2, reflecting changes it has undergone during development.

```
package com.providerA.tempsensor;
public interface TempSensSvc {
    public double getCurrentTemp();
    public double getMaxTemp();
}
-----
Export-Package: com.providerA.tempsensor;
    version="1.0.0"
```

Listing 1: Thermometer bundle revision 1

The weather station bundle imports the `tempsensor` package so it can access the thermometer service interface. It then calls its `getCurrentTemp()` and `getMaxTemp()` methods to compute some aggregate results. The version deployed in a target framework was build against the 1.0.0 version of the `tempsensor` package.

```
public interface TempSensSvc {
    public double getCurrentTemp();
}
-----
Export-Package: com.providerA.tempsensor;
    version="1.1.0"    <-- problem here
```

Listing 2: Thermometer bundle revision 2

Now when the thermometer bundle is to be updated to revision 2, its meta-data – here the `version` attribute in the `Export-Package` header – make the framework erroneously believe it is backwards compatible (following OSGi standard policies, change in the minor version number indicates compatible change). The update and subsequent package wiring process therefore proceeds smoothly and the weather station is bound to the new version of `tempsensor`.

Obviously, interface of the second revision's service does not match the expectations of the weather station bundle since the `getMaxTemp()` method is missing. The framework however has no means of detecting this problem as the package version number is incorrect. The effect is that a `java.lang.NoSuchMethodError` or similar is thrown by the weather station bundle code.

Setting correct meta-data is therefore crucial for reliable functioning of component applications. A human mistake or negligence can nevertheless result in false positives, i.e. version identifiers claiming compatibility when in fact the underlying code is incompatible. Situations like the one described above result easily (even if not very often) and the consequences can be serious for a fielded application.

3 The Current State of the Art

This section surveys research work and current technical standards related to component updates and their correctness. There are a number of foundational works that set the scene for component compatibility. Among others, [18] provides the terminology, defining (signature) compatibility as the ability to correctly exchange messages and data between client and supplier at the syntactical level.

3.1 Feature binding and compatibility

The OSGi framework [11] offers several mechanisms to resolve bundle dependencies. The core framework's resolution process handles the above-mentioned package dependencies enhanced by version id/range matching. The declarative services specification [10, Chapter 112] stemming from the Service Binder research [5] addresses declarative service dependency resolution, including runtime changes. The OSGi Bundle Repository [1] helps to resolve the cascading package dependency problem, by providing a transitive closure of interdependent bundles given a "root" bundle.

Declarative services use two policies with respect to handling unsatisfied dependencies. The standard "static policy" means that client components are deactivated after a provided service is unregistered. The "dynamic policy" means the client is notified about the service change which gives it a chance to react in a flexible manner. Dynamic policy can probably use service versioning (through service properties) but this is not explicitly mentioned in the specification.

Neither the core OSGi framework nor the extension services handle type mismatches however since they do not analyze package or service interface content. Thus when the package or service reference obtained does not match the compiled-in class expected, runtime error occurs. This can happen if (and only if) there is an incompatibility in the update version and it is not correctly marked by version identifiers (for exported packages or services).

In a recent OSGi-related development, the iPOJO concept [6] uses a similar concept of an explicit service dependency handler attached to the component. This provides both explicit service declarations and the possibility of adding (in principle) compatibility controls if needed.

The formal assurance of component compatibility is the subject of several research works. Zenger for example [19] proposes an approach that guarantees safe updates using a calculus of modification operations. While this method is formally sound it is largely inapplicable to industrial component models because it is limited to a subset of the Java language.

McCamant et al [8] define compatibility based on observed (not declared) behaviour. This is certainly more precise than pure specification-based type reconstruction used in our work. On the other hand it is much more difficult to obtain the real interaction protocol type of a component.

Lastly we note that the use of version identifiers to denote software compatibility has a long tradition in software engineering [12, 13, 15, 11]. The common approach is to provide a three-level version number scheme ("major.minor.micro") where change in the first component denotes client-affecting incompatibility.

3.2 Obtaining feature type information

In order to check feature types for type equality or subtype relation, it is first necessary to discover and have access to the type information. As Alvaro et al note [2] in practice, very little information about a black-box component is sometimes available to its (end)user. This includes the type information about surface features of OSGi bundles as described in a work related to the presented one [17].

The problems resulting from undeclared information about component services are discussed e.g. in [7]. The authors conclude that insufficient declarations result in unnecessary complex wiring code. Consequently the prototype Service Binder implementation [5] mentioned above is proposed which uses XML meta-data to explicitly declare service types and manage their dependencies.

Analogously the OSGi bundle repository provides a descriptive format and a discovery mechanism to learn about bundle capabilities and requirements. However, only type names are declared (introspection or other means are needed to obtain full type information) and the repository does not cater for installed bundles.

4 Type-safe Updates: Concepts and OSGi Realization

In the first two sections of this paper, we have explained the problems caused by relying on potentially incorrect meta-data in component binding and updates in particular. Let us now describe how to improve the situation by employing type-based checks in this process. We deal with the general concepts first and then present the particular realization for the OSGi framework.

The principle is fairly straightforward and has been described earlier [3]. Type-safe updates are based on the fact that to ensure correct application functionality, the type of the provided (exported) feature must be equal to or subtype of the required (imported) feature's type. Since the information about type compatibility in component meta-data need not be present can be unreliable, we want to verify this relation directly on the types in question during the feature binding process. Such verification is used instead of, or in addition to, the component framework's standard checks.

Now even though the type-based checks are a sufficient guarantee of syntax-level compatibility and consequently runtime safety of the application (i.e. they ensure no typecast and "no such method" exceptions can occur), we prefer the "in addition to" approach. This way we honour the other possible reasons for marking the importer-exporter pair as incompatible, for example due to extra-functional properties mismatch or for business reasons (deprecating old versions of a service). With OSGi, it is moreover very sensible to build upon the already existing controls, namely package versioning, bundle resolution and service binding.

We still have two options with respect to how the type-based verification is placed alongside these framework checks:

1. *Before framework.* Type checking works as first sentry which prevents further (needless) checks on bundles which exhibit type mismatch.
2. *After framework.* Type compatibility verification is run when framework controls

have passed, as a last pre-flight check before the bundle is declared as resolved and features are bound.

Both approaches will ensure type safety and both have their merits, so the concrete choice depends on other factors. In the following subsection we discuss the design chosen for two OSGi framework implementations.

4.1 Realization for OSGi³

In the particular case of the OSGi framework, the processes of updating a bundle and binding corresponding feature pairs are handled by the framework core and standardized services as described in Section 1.1. Also, the overall architectural style of OSGi suggests, and the advanced possibilities of the framework enable, that extensions are implemented as services exported by bundles installed in user space.

We therefore designed the verification with the following architectural goals in mind:

- No framework modifications. It should be possible to add the verification without recompiling framework source.
- User space. The verification should be implemented as a bundle providing services, so that it can be simply installed in a framework.
- Maximize reuse. Implementation should use available standard framework features as much as practically possible.

The final design uses the *Before framework* approach and has been successfully realized for the Knopflerfish and Apache Felix framework implementations; the Equinox was also studied. It employs an UpdateController bundle which drives the process plus a user interface bundle as its front-end.

To check new bundle version's compatibility with the currently installed one, it first retrieves the JAR archive of the current one from framework's bundle cache and downloads the new version's JAR from the update location. (The location is determined in the standard way, using `Update-Location` manifest header if set or calling the `Bundle.getLocation()` method; cf. [11, Section 6.1.4].)

Next, the types of surface features of both bundles are reconstructed [17] and compared [3, 16]. If the bundle comparator signals problems (the new version is not a subtype or equal to the current one) the UpdateController reports them to the user and stops the process. Otherwise, it calls standard `Bundle.update()` method on the current bundle with the new version as parameter.

The implementation therefore ensures that the existing clients will be able to safely bind to the update bundle. Figure 1 shows the user interface of the checking process in the Knopflerfish implementation.

³The author would like to thank his student Jaroslav Bauml for the work on the Knopflerfish and Felix implementations.

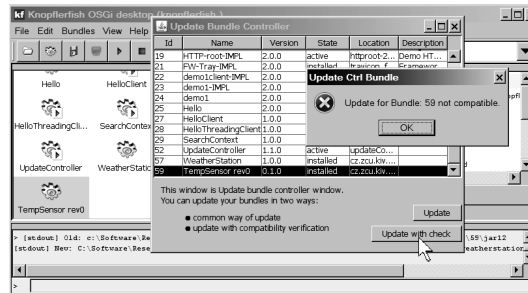


Figure 1: User interface of UpdateChecker

4.2 Discussion

Let us now describe several design considerations and discuss the weaker points of this design of type-based OSGi update verification. The update process is discussed first, followed by the properties of the type-based verification.

The original, and in hindsight uninformed, intent was to use the *After framework* approach. The idea was to intercept the process using synchronized event listeners bound to the appropriate bundle event (fired after the current version has been stopped and unresolved, before the new one is to be installed). Inside the listener event handler, the compatibility checks would be run.

Unfortunately the design OSGi update process would require framework core changes to enable this approach (see Section 5 below for details). While this is technically possible to implement, at least in the case of open source frameworks used in the experiment, it was a no-option under our architectural goals.

The resulting implementation of the type-based update checks therefore suffers from the inherent weakness of the *Before framework* approach: it does not harness the framework checks, unnecessarily repeating them in case the bundle meta-data correctly indicate incompatibility.

The hardest task of the type-safe bundle updates is obtaining a faithful representation for all component’s feature types. We use introspection on the bundle JAR package combined with bytecode analysis. As described below, the key problem is discovering service declarations; the resulting bundle comparison therefore does not verify service compatibility in a reliable manner.

Furthermore, the bundle comparator used in the verification does not implement complete type-theoretic notion of subtyping [4]. Java language uses static type binding in which methods are considered compatible only if their signatures are equal, including referenced types. Therefore we reduce the cause of subtyping on interfaces to method insertion/removal. For more detailed description of this issue see [3].

Finally, the goal of minimizing framework modifications required to create a separate user interface to the update checking extension (a new shell command and/or GUI widget). In other words, the verification is not accessed completely transparently via existing means e.g. an `update` command. On the other hand, we do not consider this a major issue – the user has the choice to run the standard “plain” update or the enhanced

one, and even modifying the shell bundle would be acceptable if needed.

5 Experiences with OSGi Updates

The implementation of the type-based compatibility verification for OSGi bundles uncovered several issues concerning the framework design and capabilities. In this section we describe the key ones, concerning especially bundle representation and the update process.

5.1 Bundle type representation

In order to correctly evaluate bundle compatibility, a complete representation of its surface features is needed. This entails the information about the existence of a feature, its name, language type, and any additional information (e.g. access constraints, version identification). In the case of black-box software components the ease of feature discovery depends on the component framework's design. For instance, in the case of CORBA Component Model [9] the type information repository can be queried.

In OSGi, the task is more difficult⁴. The representation is built starting from component meta-data (the manifest file) where (some of) the component surface features are declared. Their type information is obtained using an analysis of the component distribution package.

The first issue is discovering the very existence of surface features. While it is easy to find the names of exported and imported packages (and consequently types, that is classes and interfaces), services, perhaps the most important of bundle features, are very difficult to discover. They need not be declared in bundle metadata (cf. [11, Paragraph 3.2.1.20]) and their creation or lookup is usually buried inside bundle's methods. Unless sophisticated bytecode analysis is applied, the information that a bundle exports or uses some services cannot be reconstructed from the distribution JAR package.

Even for the known features, obtaining type information can be a problem in the case of imported features. The respective classes reside in packages which are usually not included with the evaluated bundle. Introspecting their contents is therefore impossible, and we have to resort to reconstructing the imported types by bytecode analysis [17] which is costly and has unreliable results. The access to execution environment's libraries and bundles can help but this approach is not applicable to stand-alone compatibility verification. We therefore see the methods for reconstructing type information of imported features as an open issue for further research.

5.2 Accessing current bundles

One of the key prerequisites of the type-based update checks in OSGi is the ability to access the distribution package of the current bundle in framework's bundle repository/cache. Most component frameworks provide standard means (APIs) of getting the meta-data and lifecycle control interfaces of the running components (e.g. the *home*

⁴Remember that we work on the distribution format of components, without access to the source code.

interface of Enterprise JavaBeans or `BundleContext` in OSGi). Obtaining the original *distribution package* of a given component is a different task however.

In OSGi there is no standardized API or service for this purpose, probably because the need for such functionality arises very rarely in normal use of the framework. All the framework implementations studied fortunately use a reasonably organized filesystem cache of installed bundles. Felix has the cleanest design of internal classes accessing the cache (the `BundleCache` class) which is ideal for these purposes. However, none of these internal classes are exported by the framework core bundle and thus cannot be used by the update checking bundle.

Our solution was therefore to add a small facade to the framework consisting of a single interface and an implementation class. It provides methods to obtain a stream with the JAR package of the bundle corresponding to a bundle identifier provided as parameter. (Due to the core bundle export limitations the class unfortunately has to duplicate parts of the cache access code in both the Knopflerfish and Apache Felix cases.) This design tries to achieve a reasonable compromise between the architectural goals and the constraints of the OSGi framework specification.

What we would have liked to see, instead of creating this facade API, is that such interface was available as a standard package or service. Then, obtaining to distribution packages could in principle be similar to e.g. accessing the bundle repository [1]. (Obviously, this would need to be combined with appropriate security mechanisms to prevent unauthorized manipulation by user-space bundles.)

5.3 Customizing the update process

As noted above, the OSGi specification describes precisely some aspects of the update process but leaves several open questions. From the point of view of this work, the most important one is the inability to intercept and customize the update process.

The specification does not provide any means of detecting that an update is on the way, in the first place. The standard mechanism of firing bundle events on various points in bundle lifecycle does not include events that would trace the update process.

What we would envisage are an `UPDATING` event fired when an update is started (similarly to `STARTING` etc.) and/or an `UPDATE_STOPPED` event fired after the current version has been stopped for update. Presumably, with such events and the existing synchronous listeners, one could plug in various user-defined actions into the update process.

Secondly, the update process cannot be aborted in case something goes wrong. Currently the only problem addressed is when the update version cannot be installed (the process is aborted and `BundleException` raised, per section 6.1.4 of the specification).

The synchronous bundle listeners have no direct means of signalling a problem to the caller, so even if they were used with the envisaged update events it would be cumbersome to stop the process. The `BundleActivator` interface which customizes bundle starting and stopping does not know whether the bundle is undergoing an update. It cannot therefore be used to insert the verification steps we need.

A plausible remedy would be the ability to “veto” a bundle state transition, in a way similar to JavaBeans `VetoableChangeListener` mechanism. In the update

process case, such listener would throw an exception if it detects a compatibility problem during update. The modified update algorithm could then be aborted or rolled back accordingly.

6 Conclusions and Future Work

The OSGi component framework provides a simple yet feature-rich component model with solid handling of component lifecycle. This paper addresses a potential weak point in the framework, where incorrect feature version identifier can cause run-time exceptions to be raised despite the rigorous bundle resolving process.

We propose the following points as the main contributions of this work. Firstly, we have shown how a formally sound verification of feature compatibility (using subtype checks) can be integrated in the update process. Secondly, the experiences from the implementation in two open-source OSGi framework implementations lead us to suggest potential enhancements of the framework including a finer-grained visibility into the bundle update process. In fact, the type-based checks are applicable to the bundle resolution process in general as hinted in the paper.

There are several outstanding open issues and areas of potential follow-up work. The implementation currently uses only bundle manifest meta-data; it would be beneficial to analyse also the descriptions available for declarative services and in the bundle repository. Considering the way OSGi handles bindings to current version of the bundle, the update checks should implement the notion of contextual compatibility suggested in an earlier work of the author. Lastly, we noted that it is quite difficult to reconstruct type information of imported bundle features – we believe this to be an open issue for further research.

References

- [1] OSGi bundle repository. Available at <http://www2.osgi.org/Repository/HomePage>. Accessed February 2008.
- [2] A. Alvaro, E. S. de Almeida, and S. L. Meira. A software component quality model: A preliminary evaluation. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, pages 28–37. IEEE Computer Society, 2006.
- [3] P. Brada and L. Valenta. Practical verification of component substitutability using subtype relation. In *Proceedings of the 32nd Euromicro SEAA conference*, pages 38–45. IEEE Computer Society, 2006.
- [4] L. Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.
- [5] H. Cervantes and R. S. Hall. Automating service dependency management in a service-oriented component model. In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering*, 2003.
- [6] Escoffier, Hall, and Lalanda. iPOJO: An extensible service-oriented component framework. In *Proceedings of IEEE International Conference on Services Computing (SCC 2007)*, pages 474–481. IEEE Computer Society, 2007.

- [7] R. S. Hall and H. Cervantes. An OSGi implementation and experience report. In *Proceedings of the Consumer Communications and Networking Conference (CCNC 2004)*, pages 394 – 399, January 2004.
- [8] S. McCamant and M. D. Ernst. Formalizing lightweight verification of software component composition. In *Proceedings of SAVCBS 2004: Specification and Verification of Component-Based Systems*, pages 47–54, Newport Beach, CA, USA), October 2004.
- [9] Object Management Group. *CORBA Components, Version 3.0*, 2002. OMG Specification formal/02-06-65.
- [10] The OSGi Alliance. *OSGi Service Platform Service Compendium, Release 4*, July 2006. Available at <http://www.osgi.org/>.
- [11] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.1*, April 2007. Available at <http://www.osgi.org/>.
- [12] M. Peterson. *DCE: A Guide to Developing Portable Applications*. McGraw-Hill, 1995.
- [13] R. Riggs. *The Java Product Versioning Specification*. JavaSoft, 2002. Available at <http://java.sun.com/j2se/1.4.2/docs/guide/versioning/spec/versioning.html>.
- [14] C. Szyperski. *Component Software, Second Edition*. ACM Press, Addison-Wesley, 2002.
- [15] The Debian Policy Mailing List. *Debian Policy Manual*, version 3.7.2.2 edition, 2006. Available from <http://packages.debian.org/debian-policy>.
- [16] L. Valenta and P. Brada. Automated generating of OSGi component versions. In *Proceedings of ECI'06*, Kosice, Slovakia, 2006. TU Kosice.
- [17] L. Valenta and P. Brada. Modelování existujících OSGi komponent (modeling of existing OSGi components). In *Objekty 2006*. Prague, Czech Republic, 2006. In Czech language only.
- [18] A. Vallecillo, J. Hernández, and J. M. Troya. Component interoperability. Technical Report ITI-2000-37, Universidad de Málaga, Spain, July 2000.
- [19] M. Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.