Automated generating of OSGi component versions

Lukas Valenta, Premysl Brada
Department of Computer Science and Engineering
University of West Bohemia in Pilsen, Czech Republic
e-mail: {lvalenta,brada}@kiv.zcu.cz

Abstract

Software components can be found in both enterprise-wide and mobile/embedded solutions. Components are mutually linked and dependent, but encapsulated as black boxes and developed independently. They can be replaced without affecting the rest of the application. This advantage requires careful and complex compatibility checks between both component versions though, otherwise the whole application can be broken down. In present systems, versioning is often used as an instrument representing changes during software development. In this paper we discuss the versioning scheme of the OSGi Release 4 component model and propose an automated generation of component version identifiers. The mechanism is based on an automatic analysis of changes in components’ interfaces which ensures safety of OSGi component upgrades.

1. Introduction

Component-based architectures have a lot of advantages. One of the finest one is a possibility to upgrade or repair only the affected components, not the whole application. However, this convenience has its issue - the replacement for an incompatible component may break the dependencies between components as well as the consistency of entire application. Accordingly, it is necessary to insert the step of careful component substitutability check before the replacement itself.

The OSGi architecture [7] allows to define many conditions and constraints for the component relationships. The framework follows these constraints during the installation or upgrade of a component. One of the important condition is the compatibility of the version numbers of requested component interface(s). Because the semantics of the version identifiers is well defined and able to capture an incompatible change, the version constraint is a strong mechanism to prevent wiring an incompatible components.

Indeed, the correctness of those version identifiers is the important presumption. Developers must evaluate changes in component’s interface and alter the versions correctly for every published release of a component. We present a method of an automated generation of OSGi component version identifiers. They are determined on the basis of changes in component’s interfaces.

1.1 Goal and Structure of the Paper

The goals of this paper can be summarised in two points. Firstly, we would like to present the formal foundations of the method in the form of representing component specifications as types and performing subtype checks on them. Secondly, we present a practical realization of the method which is used to generate the correct version identifiers of OSGi bundles.

The following section deals with the principles of substitutability checking and determining differences between component versions. Section 3. describes the OSGi versioning schema, its relation to component differences and shows the tool for automated generating of OSGi components version. Section 4. presents an overview of related research.

2. Principles of Component Substitutability Checking

As it was mentioned in the introduction, the component substitutability check is a neces-
necessary step in the component replacement (be it an upgrade or a more general substitution). The fundamental principle of substitutability is defined in this way: a substitute component should be usable whenever the current one was expected, without the client noticing it [9]. Type systems and the subtype relation in particular are used to ensure safe substitutability in (object-oriented) programming languages: instances of type $T'$ can be bound to variables declared to be of type $T$ if $T' <: T$ (subtype) because the subtype provides a superset of features [3, 1].

2.1 Component Type Differences
Component interfaces are defined in the terms of programming language constructs (interface types, methods, etc.), therefore subtyping can similarly be used for component compatibility evaluation. Our approach says that component $B$ can replace component $A$ if $B$’s type is a subtype of $A$’s type.

To determine the subtyping relation between two types $A$ and $B$, one needs to compare the content of the types. The rules for type constructs are used recursively until primitive types are reached, rules for them are defined by enumeration (e.g. short $<: long$).

The result of comparing two types $a$ and $b$ can be described by the character of changes between them. Let us define the function $\text{diff}(a,b) : Type \times Type \rightarrow Differences$ which computes the difference between types $a$ and $b$. The returned value is one of:

- $\text{none}$ if $a = b$
- $\text{insertion}$ if $a$ is not defined but $b$ is
- $\text{specialization}$ if $b <: a$
- $\text{deletion}$ if $b$ is not defined but $a$ is
- $\text{generalization}$ if $a <: b$
- $\text{mutation}$ if $b$ contains both ins/spec and del/gen differences
- $\text{unknown}$ if $b$ cannot be compared to $a$ (e.g. due to recursive cycles)

For example, we have a Java interface called `cz.zcu.logging.Logger` (see Table 1) and want to determine the differences between the methods of its two versions. We obtain the following values. The `write()` method is not changed in version 2, its difference is therefore $\text{none}$. There is no `flush()` method in the first version, thus there is $\text{insertion}$ difference. Since `int <: long`, the last method exhibits a $\text{generalization}$ difference.

```
interface Logger { // version 1
    void write(String msg)
    int getItemCount()
}

interface Logger { // version 2
    void write(String msg)
    void flush()
    long getItemCount()
}
```

Table 1: Example interface types

In our approach, the comparison of structured types is done by combining the differences of their constituent parts. The combination of two difference values is given by their relative priority. The priorities are as follows, in increasing order: 1. $\text{none}$, 2. $\text{insertion}$, 3. $\text{deletion}$, 4. $\text{specialization}$, 5. $\text{generalization}$, 6. $\text{mutation}$, 7. $\text{unknown}$. This says that, for example, a $\text{mutation}$ has a bigger impact on the type than a $\text{deletion}$ change.

The general combination rule is as follows: Higher priority values override the lower ones, with one exception: the mutual combination of the values at the second as well as third level results in $\text{mutation}$.

For example, the values are combined as $\text{none} \oplus \text{insertion} = \text{insertion}$ or $\text{insertion} \oplus \text{generalization} = \text{mutation}$. When determining the difference between the versions of the `Logger` interface, we proceed as follows: the combined differences of the first two methods form a $\text{specialization}$, and its combination with last methods $\text{generalization}$ results in $\text{mutation}$.

3. Substitutability for OSGi Release 4
One of the appealing application areas of component substitutability checks are components (called `bundles`) running in OSGi
framework, since OSGi becomes more important and widely used in industry. OSGi bundles can be (remotely) deployed to range of devices from embedded/mobile to enterprise servers. In this section we describe how the substitutability checking and new version identifier assigning is done for component model OSGi Release 4 [7].

3.1 OSGi Versioning Schema

As an evolution of the platform from the previous release [6], OSGi Release 4 specifies versioning schema. The version identifier is assigned to particular exported packages as well as to the entire bundle. It has the generic structure \texttt{major.minor.micro} with following compatibility policy: an incompatible change is signalled by incrementing the \texttt{major} number, while a compatible one increments only the \texttt{minor} number. If there was not any change in the component’s interface, the \texttt{micro} is increased (e.g. bugfix).

Components can specify versioned dependencies, the typical import statement looks like this:

\begin{verbatim}
Bundle-Name: A
Import-Package: cz.zcu.logging;version="[1.2.5, 2)"
\end{verbatim}

The OSGi framework resolves all dependencies and constraints during bundle deployment and links importers to the right exporters. This verification (in the case of valid bundles) detects problems early, avoiding runtime errors. It is of course very important to provide correct version identifiers of the bundles and their exported packages.

If a bundle \texttt{B} exports a new version of the \texttt{cz.zcu.logging} package with an incorrect version number (e.g. 1.3.0 despite an incompatible change in one of its interfaces), bundle \texttt{A} would be successfully resolved and wired to this exporter (\texttt{B}) upon deployment. The incompatibility would surface only at runtime when an attempt is made to access the mentioned interface. The problem is obviously caused by the manual assignment of version numbers.

This hand-made activity can produce human mistakes and it can be automated. In the next section we present the method for automated generation of bundle interface version identifiers. The mechanism is based on an analysis of changes in bundle’s interfaces which ensures safety of OSGi component upgrades.

3.2 Differences and OSGi version related

The knowledge of difference between two subsequent versions of an interface part is sufficient to determine the new version identifier of that part. Let \(d = \text{Diff}(R_{i-1}, R_i)\) be the difference between two consecutive revisions, \(V_{old} = \text{maj}_{old} \cdot \text{min}_{old} \cdot \text{mic}_{old}\). Then the new version identifier \(V_{new} = \text{maj}_{new} \cdot \text{min}_{new} \cdot \text{mic}_{new}\) is defined by the rules in Table 2.

<table>
<thead>
<tr>
<th>\text{Diff}(R_{i-1}, R_i)</th>
<th>\text{maj}_{new}</th>
<th>\text{min}_{new}</th>
<th>\text{mic}_{new}</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>\text{maj}_{old}</td>
<td>\text{min}_{old}</td>
<td>\text{mic}_{old} + 1</td>
</tr>
<tr>
<td>specialization, insertion</td>
<td>\text{maj}_{old}</td>
<td>\text{min}_{old} + 1</td>
<td>0</td>
</tr>
<tr>
<td>deletion, generalization, mutation</td>
<td>\text{maj}_{old} + 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>unknown</td>
<td>unknown</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Derivation of the new version identifier

For example, assume that we have a \texttt{LogService} component which exports the package \texttt{cz.zcu.logging} with the only one interface \texttt{Logger} from Table 1, and that version 1 of the interface has version identifier 1.2.1. If the interface changes to its version 2 for a new release of the component, the \texttt{mutation} difference between the interface versions signals an incompatible change. Therefore, the new release of package \texttt{cz.zcu.logging} will have 2.0.0 as its version number according to line 3 of the table. If however there was only a \texttt{specialization} change in the \texttt{Logger} interface, that release would be numbered 1.3.0 (line 2 of the table).

Java package is the unit of export/import in OSGi, it contains classes and interfaces in
general. To compute the difference of a particular exported package, one must compare every public class and interface contained in the package with its older version. The combination of gained differences results in the difference of the whole Java package and thus to its new version identifier.

3.3 Comparing the entire OSGi component

In the previous chapters we presented the method for comparing exported packages and determining their new version identifier. Now we show how to determine the new version of the entire bundle.

Firstly, it is needed to discover all elements of bundle’s interface. Based on the OSGi specification [7] we have defined subtyping rules for OSGi bundles that cover all of the elements of their interface. The rules come in two complementary sets: one covers the functionality features, the other one the non-functional tags attached to the whole component.

Tables 3 and 4 show the rules for features and tags. Every feature has either provided (P) or required (R) role. Roles must be taken in account when differences of each features are merged together. Differences of required features have to be reversed before they are merged with provided ones – the formalism that captures the effect of these roles in type theory is contravariance [3].

<table>
<thead>
<tr>
<th>Feature</th>
<th>Role</th>
<th>Type compared</th>
</tr>
</thead>
<tbody>
<tr>
<td>export types</td>
<td>P</td>
<td>Java class, tags</td>
</tr>
<tr>
<td>import types</td>
<td>R</td>
<td>Java class, tags</td>
</tr>
<tr>
<td>native code</td>
<td>R</td>
<td>List, tags</td>
</tr>
<tr>
<td>require bundles</td>
<td>R</td>
<td>String (bundle name), tags</td>
</tr>
<tr>
<td>required exec env</td>
<td>R</td>
<td>List</td>
</tr>
</tbody>
</table>

Table 3: OSGi feature subtyping rules

Additional tags are attached to the component itself as well as to its particular features. Tags have to be compared using rules defined in the table 4.

The Type compared column describes what is compared - Java class, String, Map, List of Strings or other representation.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Belongs to</th>
<th>Type compared</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbolic name</td>
<td>import types</td>
<td>String</td>
</tr>
<tr>
<td>bundle version</td>
<td>import types, require bundles</td>
<td>Version interval</td>
</tr>
<tr>
<td>language</td>
<td>native code</td>
<td>List</td>
</tr>
<tr>
<td>os name</td>
<td>native code</td>
<td>List</td>
</tr>
<tr>
<td>os version</td>
<td>native code</td>
<td>List</td>
</tr>
<tr>
<td>parameters</td>
<td>export types, import types</td>
<td>Map</td>
</tr>
<tr>
<td>processor</td>
<td>native code</td>
<td>List</td>
</tr>
<tr>
<td>resolution</td>
<td>import types, require bundles</td>
<td>Enumeration</td>
</tr>
<tr>
<td>symbolic name</td>
<td>bundle</td>
<td>String</td>
</tr>
<tr>
<td>version range</td>
<td>import types</td>
<td>Version interval</td>
</tr>
<tr>
<td>kind</td>
<td>import types</td>
<td>Enumeration</td>
</tr>
</tbody>
</table>

Table 4: OSGi tag subtyping rules

3.4 Algorithm of comparison

In this section we present the complete algorithm of OSGi component comparison described in previous chapters. Two subsequent versions of component $A_1$ and $A_2$ are used as input. The goal is to determine version identifiers of $A_2$ – versions of exported packages and of the whole bundle. The algorithm is as follows:

1. Compare the same provided features (and their possible tags) of $A_1$ and $A_2$ according to the Type compared column in Table 3, and save the differences found.

   (a) Take acquired differences of export types feature and combine those referring to the same Java packages. As a result of this you gain the difference of every package exported by component.

   (b) Generate the new version identifiers of each of those packages according to Table 2.

2. Compare the required features in the same way.

3. Compare equivalent component-wide tags according to Table 4, and save the differences.

4. Compute the overall difference:
(a) Combine all differences of provided features and tags according to Section 2.1.
(b) Combine all differences of required features in the same way
(c) The difference of the components is given by the diff value combination and contravariance rules.

5. Generate the new version identifier of the whole component $A_2$ according to Table 2.

The overall result of the check is a single difference value at the component level. If the value is in the \{none, insertion, specialization\} set then $A_2$ is backwards compatible with $A_1$, and the major part of version identifier stays unchanged. In any other case, an incompatible change has happened and the new major version has to be introduced.

3.5 Implementation of OSGi comparison

The substitutability rules and algorithms described above can be implemented by automated checking tools. We have designed and implemented a prototype tool which allows the comparison of two versions of an OSGi component. It operates on two JAR files which each contains one version of OSGi component - the old one and the newly developed one. It reads the data of these components, performs the comparison and generates a copy of second JAR file with altered versions. They thus reflect the severity of the changes performed within bundle development.

As described in Section 3.2, the tool generates new version identifiers of the second component as well as its particular exported packages. For example, assume the first version of bundle LogService exporting package cz.zcu.logging:

```plaintext
Bundle-Name: LogService
Bundle-Version: 2.3.5
Export-Package: cz.zcu.logging;version="1.2.1"
```

Assume there was only a specialization change in exported Logger interface but the interface of the whole bundle changed more seriously and resulted to generalization change\(^1\). Than the new (generated) manifest file of the new bundle looks like this:

```plaintext
Bundle-Name: LogService
Bundle-Version: 3.0.0
Export-Package: cz.zcu.logging;version="1.3.0"
Require-Bundle: new_bundle_dependency
```

Further, a XML file with detailed comparison result is created. It covers all examined levels – from the top level (whole component) to the detailed parts of its interface (e.g. methods of Java classes).

The component data - features and tags - is obtained from two sources: (1) the JAR file manifest, and (2) the component implementation in .class files, using Java introspection mechanism. The data is stored internally in structures defined by the ENT meta-model [2] which makes it easier to analyze component interface and compare its parts.

4. Related Work

The area of component substitutability checking is well researched.

Zenger [10] describes a method that ensures safe component upgrades on the basis of a well-defined evolution mechanism supported by appropriate calculus. This would achieve the desired safe upgrades but the mechanism does not apply to current industrial frameworks. It is more likely a guide how to construct the specification of a new component model.

A closely related approach was presented by McCamant [4]. It also leads to component substitutability checking and ensuring safe upgrades, but from a quite diverse point of view. We operate on component types and behaviour gained from declared interface and on grounds of this information we determine the subtyping relation between two surveyed

\(^1\) e.g. it has a new dependency to required bundle, see Table 3
components. On the contrary, the mentioned approach takes in account an observed behaviour of component.

That method is partly similar to contextual substitutability as presented in [1]. It also takes in account only the concrete component context, but the way how it is determined differs. McCamant’s method is sensitive to quality of a test suite used to capture observed behaviour.

Versioning as an approach to ensure safe software evolution is presented for Java classes and packages as a part of Java language specification [8]. It defines versioning schema and a suggested evolution policies for Java. Although Java packages are being dynamically located and loaded all the time (and thus it is needed to ensure compatibility of them), this specification is rarely used in practice.

Very similar approach using version identifiers in the form major.minor is defined by Distributed Computing Environment (DCE) 1.1: Remote Procedure Call [5]. Version schema and rules are applied to interfaces on the server side and are used when a RPC client tries to call server’s procedure. The DCE specification defines exact rules for selecting the compatible version of services provided by server and for assigning version numbers to a newly developed interfaces.

5. Conclusion

The presented method (and its implementation) ensures that correct version numbers are assigned to an OSGi bundle. Because the framework uses these version numbers, no error caused by an incompatible component during upgrade should happen.

We investigate components in the form they are distributed – encapsulated blackboxes with no access to implementation details or source codes. We gather as much information from this form as possible and construct the representation of component’s interface. On the basis of comparison of those two representations we generate the new version identifiers. We therefore cannot capture the potential changes of internal behaviour or interface semantics because it is not a part of OSGi component interface specification.

References