University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

OSGi Component Substitutability Using Enhanced ENT Metamodel Implementation

technical report

Lukas Valenta, Premysl Brada

Technical Report No. DCSE/TR-2006-05
June, 2006

Distribution: public

Technical Report No. DCSE/TR-2006-05
June 2006

OSGi Component Substitutability Using Enhanced ENT Metamodel Implementation

Lukas Valenta, Premysl Brada

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 this paper we present and describe the implementation of the ENT metamodel and the ENT based component comparison algorithm. This algorithm is used in the practical case: OSGi Release 4 components are being compared. On the basis of the change, version identifiers of the newer component are assigned.

.

Contents

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.

If we understand the components such as the encapsulated black-boxes, we have to consider their interface only. In general, we would like to model component interface independently of the component model and/or the language it was implemented. Thus we should use some component metamodel such as UML or ENT [3]. If the metamodel is capable to capture all relevant and important information about the component interface, we can define the subtyping rules on the metamodel level and perform the substitutability check on the level of metamodel representation of the component interface.

The subtyping rules form the general algorithm of component comparison. In this paper we present the implementation of the ENT metamodel including the component comparison algorithm.

To show the practice use of the method and algorithm, the case study using the OSGi architecture follows.

The OSGi architecture [10] 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 goal of this paper can be sumarised in two points: (1) description of the ENT metamodel implementation and its novelties and (2) the case study of the OSGi component comparison.

The first chapter focuses on the OSGi Service Platform, it contains a short introduction to the OSGi common properties, component model, versioning schema and is concluded by the description of two projects using OSGi. Second chapter deals with the principles of component substitutability checking. Next chapter is the core of this paper, it descibes the ENT metamodel [3] prototype implementation and its enhancements since the previous version.

Next chapter contains the case study on the OSGi component model. It presents the method of comparing OSGi bundles and the relation with the versioning schema. The paper is concluded with the review of the related work.

2 The OSGi Service Platform Release 4

The OSGi Service Platform is open, common architecture for service providers, developers, software vendors, gateway operators and equipment vendors to develop, deploy and manage services in a coordinated fashion. It enables an entirely new category of smart devices due to its flexible and managed deployment of services. OSGi specifications target settop boxes, service gateways, cable modems, consumer electronics, PCs, industrial computers, cars, mobile phones, and more. Devices that implement the OSGi specifications will enable service providers to to deliver differentiated and valuable services over their networks[10].

In this paper we deal with the fourth release of the OSGi specification, it is backward compatible with all older versions and presents many new features.

The core of the OSGi Service Platform is formed by the Framework (in other component models is usually called *container*). It provides a general-purpose, secure, and managed Java framework that supports the deployment of extensible and downloadable applications known as *bundles*.

OSGi-compliant devices can download and install OSGi bundles, and remove them when they are no longer required. The Framework manages the installation and update of bundles in an OSGi environment in a dynamic and scalable fashion. To achieve this, it manages the dependencies between bundles and services in detail.

It provides the bundle developer with the resources necessary to take advantage of Javas platform independence and dynamic code-loading capability in order to easily develop services for small-memory devices that can be deployed on a large scale.

2.1 Common Properties of OSGi

- OSGi Platform is intended for mobile/embedded devices $\rightarrow$ runs in the Java Micro Edition (J2ME) [14].
- It defines simple component model. Every component (called *bundle*) is an application which can cooperate with another components. The framework runs in the single Java Virtual Machine.
- Bundles can export and import services to/from other bundles.
- OSGi Platform does not support transparent component distribution over many JVM like more complex component models such as EJB or CCM [13, 15]. But it is possible to implement this feature manually by for example *http* and *servlet* services.
- OSGi Framework offers the well defined set of common services: logging, xml parser, http, servlets. . . .
- The latest release 4 supports versioning of the whole bundles and their individual interfaces too. Bundle dependencies are resolved with respect to these versions.

2.2 Bundles

The Framework defines a unit of modularization, called a bundle. A bundle is comprised of Java classes and other resources, which together can provide functions to end users. Bundles can share Java packages among an exporter bundle and an importer bundle in a well-defined way.

Bundle is deployed as a Java archive (JAR) file. This file contains

- the resources necessary to provide some functionality. These resources may be class files for the Java programming language, as well as other data such as HTML files, help files, icons, and so on.
- a manifest file describing the contents of the JAR file and providing information about the bundle. This file uses headers to specify information that the Framework needs to install correctly and activate a bundle.

Once a bundle is started, its functionality is provided and services are exposed to other bundles installed in the OSGi Service Platform.

2.3 OSGi Versioning Schema

As an evolution of the platform from the previous release [9], 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 *major.minor.micro* with following compatibility policy: an incompatible change is signalled by incrementing the *major* number, while a compatible one increments only the *minor* number. If there was not any change in the component's interface, the *micro* is increased (e.g. bugfix).

Components can specify versioned dependencies. The example of package export follows:

```
Bundle-Name: B
Export-Package: cz.zcu.logging;version=1.3.0
```

This means that bundle exports all public classes and interfaces from Java package `cz.zcu.logging`. The single export can contain attributes such as *version*, which specifies the version of exported package.

Version constraint is a mechanism whereby an import definition can declare a precise version or a version range for matching an export definition. There can be many different versions of the same package in the framework. Upon the requirements of the importer, the most sufficient version of exported package must be chosen. The typical import statement looks like this:

```
Bundle-Name: A
Import-Package:
  cz.zcu.logging;version="[1.2.5, 2)"
```

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 B exports a new version of the `cz.zcu.logging` package with an incorrect version number (e.g. 1.3.0 despite an incompatible change in one of its interfaces), bundle A would be sucessfully resolved and wired to this exporter (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 this paper 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.

2.4 Default OSGi Services

OSGI Platform Specification defines a set of default services which must be present in every OSGi Platform implementation. As all applications and services in OSGi, default services are implemented as bundles exporting related packages. The most important services are

Log Service The Log Service provides a general purpose message logger for the OSGi Service Platform.

Http Service An OSGi Service Platform normally provides users with access to services on the Internet and other networks. This access allows users to remotely retrieve information from, and send control to, services in an OSGi Service Platform using a standard web browser.

Preferences Service Many bundles need to save some data persistently – in other words, the data is required to survive the stopping and restarting of the bundle, Framework and OSGi Service Platform.

XML Parser Service The Extensible Markup Language (XML) has become a popular method of describing data. As more bundles use XML to describe their data, a common XML Parser becomes necessary in an embedded environment in order to reduce the need for space.

Device Access A Service Platform is a meeting point for services and devices from many different vendors: a meeting point where users add and cancel service subscriptions, newly installed services find their corresponding input and output devices, and device drivers connect to their hardware.

Beyond the mentioned ones the specification defines many additional standard and optional services. Every OSGi implementation must contain them.

2.5 Example of OSGi Application: Eclipse Platform

Example of OSGi application is the Eclipse Platform [11]. It is a platform that allows integration of tools for development of various application types, it provides them the common user interface. It is based on plug-in system, it supports many operating systems and provides an os-independent layer for plug-ins. At the core of Eclipse is an architecture for dynamic discovery, loading, and running of plug-ins.

Tools that developer develops can plug into the workbench using well defined hooks called extension points. The platform itself is built in layers of plug-ins, each one defining extensions to the extension points of lower-level plug-ins, and in turn defining their own extension points for further customization. The base plug-in is "Platform runtime":

- it defines and manages plug-in and extension points model
- it dynamically discovers plug-ins and maintains information about the plug-ins and their extension points in a platform registry
- plug-ins are started up when required according to user operation of the platform.
- the runtime is implemented using the OSGi framework (plug-in = OSGi bundle)

Actual version of the Eclipse Platform 3.1 uses OSGi Release 4 or more precisely its "enhanced version" - Equinox [12]. In comparison with the standart OSGi specification it contains for example some extra headers in the manifest file.

2.6 Enterprise Component Framework

Enterprise Component Framework (ECP) [5] is the appearing extension of the OSGi framework.

The outstanding flexibility of the Eclipse Platform is provided by its comprehensive component model. The Eclipse Platform component model targets tools development, tools integration and rich client applications and offers users a strong GUI orientation. Unfortunately, there is nothing similar to the Eclipse Platform available for enterprise application developers. Existing components models do not provide the flexibility for enterprise applications that the Eclipse Platform does for GUI-oriented applications

A majority of the flexibility the Eclipse Platform inherited from the OSGi Service Platform. Similarly, the Enterprise Component Framework should also be based on and extend the OSGi Service Platform. The Equinox along with Eclipse Core will serve as a container for the enterprise components.

Thsi project is only at the beginning but it offers some interesting features. One of them is the specification of dynamic behaviour of the components. The components should adapt to the dynamic availability (the arrival or departure) of the services or other components they are using. This feature makes this project important from our point of view and thus we will monitor the project progress.

3 Principles of Component Substitutability Checking

As it was mentioned in the introduction, the component substitutability check is a 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* [19]. 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 [6, 2].

3.1 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 $diff(a, b) : Type \times Type \rightarrow Differences$ which computes the difference between types a and b. The returned value is one of:

none	if $a = b$
insertion	if a is not defined but b is
specialization	if $b <: a$
deletion	if b is not defined but a is
generalization	if $a <: b$
mutation	if b contains both $ins/spec$ and del/gen differences
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 *none*. There is no `flush()` method in the first version, thus there is *insertion* difference. Since *int* <: *long*, the last method exhibits a *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 (e.g. a whole Java interface) is done by combining the differences of their constituent parts (the operations of the interface). Thus, for level n of the type structure, we compute its difference value as follows: first obtain the difference values for all constitutent parts at the $n-1$ level, then combine the values into a single one.

Clearly, the effect of differences on the type's substitutability varies. We express this effect by weight measures assigned to the values, as follows: 1. *none*, 2. *insertion*, *deletion*, 3. *specialization*, *generalization*, 4. *mutation*, 5. *unknown*. This says that, for example, a *mutation* has a bigger impact on the type than a *deletion* change.

The difference value for level n of the type structure is then obtained as follows:

1. Obtain the difference values for all constitutent parts at the $n-1$ level.
2. When combining values for type parts playing contravariant roles within the level n (e.g. the provided and required interfaces of a component), invert the difference values of the parts with the "required" role: $ins = \overline{del}$, $spec = \overline{gen}$, other values stay the same.
3. Combine the values into a single one, using the following algorithm:
   (a) Sort the differences at the $n-1$ level by their weights.
   (b) If the highest weight is from the set $\{1, 4, 5\}$, use the corresponding difference for the level n value.
   (c) Else, if there are differences with weights both 2 and 3 in the set, use Table 2 to determine the resulting difference.
   (d) Else (no combination of weights 2 and 3), use the difference corresponding to the highest weight for the level n value.

	ins	*del*	*spec*	*gen*
ins	ins	mut	spec	mut
del		del	mut	gen
spec			spec	mut
gen				gen

Table 2: Combination of contravariant difference values

3.2 Component Substitutability Defined by Differences

When we want to ensure that component B can safely replace component A, we need to determine that $B <: A$. In this case, we work at the level of the whole component where the provided and required roles of its interface parts (e.g. the types in Export-Package vs. Import-Package declarations of an OSGi bundle or the business interface vs. `ejb-ref` business references of an Enterprise JavaBean [13]) have to be considered.

The formalism that captures the effect of these roles in type theory is contravariance [6]: B is a subtype of A only if the provided part of B is a superset and the required part is a subset of corresponding A's parts. This can be re-phrased as the (obvious) requirement that B provides at least the same functionality and requires at most as much as A did.

Using the difference values we therefore say that B is a (strict) substitute for A if and only if

- $diff(A_{prov}, B_{prov}) \in \{none, insertion, specialization\}$, and
- $diff(A_{req}, B_{req}) \in \{none, deletion, generalization\}$

In simple words, functionality must not change or can only be added at the client-side of components interface. On the other hand, reduction is the only allowed change at the side of components dependencies.

In the practice one must consider the nature of used programming language. There is the difference between statically and dynamically linked languages. The full description of this problem can be found at [4].

4 ENT Metamodel Implementation Enhancements

At this place we continue with the work on ENT metamodel [3] prototype implementation. It started in the master thesis [16], where we dealed with Enterprise JavaBeans [13] component modelling. The goal of that thesis is to design and implement the program creating the ENT representation of existing EJB components. The ENT representation is complex data structure, its exact format is defined individually for particular component model[1].

The core of the program is the implementation of common ENT metamodel. It can be divided to two parts: The definition part specifies the particular component model, it defines traits and tags (their names, relationships, possible values...) of the component interfaces. Second part represents the interface of particular component, so it consists of elements and tags. This core forms the abstract API which can be used to model any component of any component model.

There stand the model-specific routines above the program core - definitions of concrete component model and tools used for reading component interface information.

The documentation for the first version of the implementation is available in the mentioned master thesis [16]. At this place we describe the changes and innovations.

Note: Every Java class and package names does not include the common prefix `cz.zcu.kiv.ent`.

4.1 Definition Part of ENT Metamodel

The theoretic definition of ENT metamodel structure has changed a little since the first version of implementation. The only enhancement appeared in the tag definition - the metatype of the tag was added. The most important (and only implementation) change is the use of new Java 5.0 language features (generics, enumeration types and new constructs).

4.2 ENT Based Comparison

The most important novelty is the implementantion of ENT based component comparison. The generic rules for strict component comparison are defined in [2],

[1]It is based on the study of that model, its features and properties

Chapter 4 and 5. The idea of the method was already described in Section 3.1 on the Page 10. All the classes representing some level of ENT component representation (element, tag, trait, ...) now implement the interface `ENTComparable`. They thus have the ability to be compared with the appropriate counterpart.

At this place the work integrate the Java class comparing code designed and implemented by Pavel Stuna [17].

- `class Diff`
  This class represents the difference between two objects (classes, interfaces, functions, tags, traits, components, ...). As described in section 3.1, it can be one value or combination of the following values:

$$\begin{aligned} Diff \in \{&None, Insertion, Specialization, Deletion,\\ &Generalization, Mutation, Unknown\}\end{aligned}$$

- `interface ENTComparable`
  This interface represents any object which can be compared with the another one according to ENT comparison rules. It has one significant method

```
public interface ENTComparable {

    public DiffResultPart compareENT(ENTComparable object);

}
```

  This method compares *this* and referenced *object* and returns result of this comparison - instance of `DiffResultPart` interface (it will be described later).

- `interface ENTComparableByName`
  Interface represents named objects. This objects must have the same name in order to be possible to compare them. If their names are not equal, comparison should return *Mutation*.

```
public interface ENTComparableByName extends ENTComparable {

    public boolean hasSameName(ENTComparableByName object);

}
```

  Comparable by name are for example instances of classes `Component`. Two components must take the same name in order to be comparable.

The ENT metamodel represents the component interface as the set of traits and the tags. When one want to compare two component ENT representations, it is needed to start at the bottommost level of the tree structure and propagate the results to the higher levels. After it, the difference of the whole component is stored at the highest level of the structure (see the diagram at Figure 1).

In this case, one will get simple result - the difference of the whole component (e.g. *specialization*). Relevant information would be lost. In practice it is important to know which changes happened in the component interface and at which part of it. Because this information is known and used during the comparison process, the only task is to store them to some data structure. This structure is represented by the Java interface `DiffResultPart`.

This interface represents result of component specification part comparison. It consists of the difference value, references to objects being compared, additional information and the results of the subordinate parts - so the result parts construct nested tree structure. Because this is designed as universal structure, every level has its name (f.e. *component*, *exports*, *tags*, ...). There is no restriction for particular component-comparison algorithms to smooth the level of detail anymore. In the case of comparing for example Java classes, additional levels like *class*, *method* or *attribute* can be added to the result hierarchy.

This interface therefore specifies access to following attributes:

- `ENTComparable object1, object2;` Objects (specification parts) being compared. For example two instances of interface elements.
- `Diff diff;` Difference between compared objects
- `DiffInfo info;` Additional textual information

The `DiffResultPart` interface extends the Java standart interface `Collection`, it is therefore collection of `DiffResultPart`*s* at lower level.

The general comparison rule at any level of ENT representation is as follows:

1. Compare the actual part to its appropriate counterpart (in the second ENT structure) and store the result to the new instance of the `DiffResultPart` interface.
2. Gather the difference results of all subordinate parts[2]. This is done by calling the `compareENT()` method of all underlaying parts. Attach these results to the mentioned new instance of `DiffResultPart` (= build the tree structure).

[2]parts on the underlaying level

3. Combine these differences to the single "difference of the actual part" using the combination rules defined at Section 3.1 on the Page 10. One must take in account the meaning of each difference - for example, differences of required traits have to be reversed before they are merged with provided ones and with the tags – the formalism that captures the effect of these roles in type theory is contravariance [6].

4. Return the resulting difference.

The result of the component comparison is therefore a `DiffResultPart` structure, which contains information about the differences at every "level of detail" of the component representation. This structure can be stored as a XML file, its format is specified at Appendix B.1 of this paper.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE entdiff SYSTEM "entdiff.dtd">

<entdiff>
  <component ctype="OSGi_R4" model="OSGi_R4">
    <name>Log Service</name>
    <part name="" level="component">
      <diff>Mutation</diff>
      <part name="Exports" level="category">
        <diff>Mutation</diff>
        <part name="export_types" level="trait">
          <diff>Mutation</diff>
          <part name="cz.zcu.Logger" level="class">
            <diff>Mutation</diff>
            <part name="flush()" level="method">
              <diff>Insertion</diff>
            </part>
            <part name="getItemCount()" level="method">
              <diff>Generalization</diff>
            </part>
          </part>
        </part>
      <part name="Needs" level="category">
        <diff>None</diff>
      </part>
      </part>
      <part name="" level="tags">
        <diff>None</diff>
      </part>
    </part>
  </component>
</entdiff>
```

This "complex" result allows identifying the elements of component interface, that are responsible for the incompatibility of two component versions, for example. As you can see in the XML file above, the information is much detailed - differences at the Java method level are detected and presented. Information may be valuable when one wants to adapt the new incompatible component so that

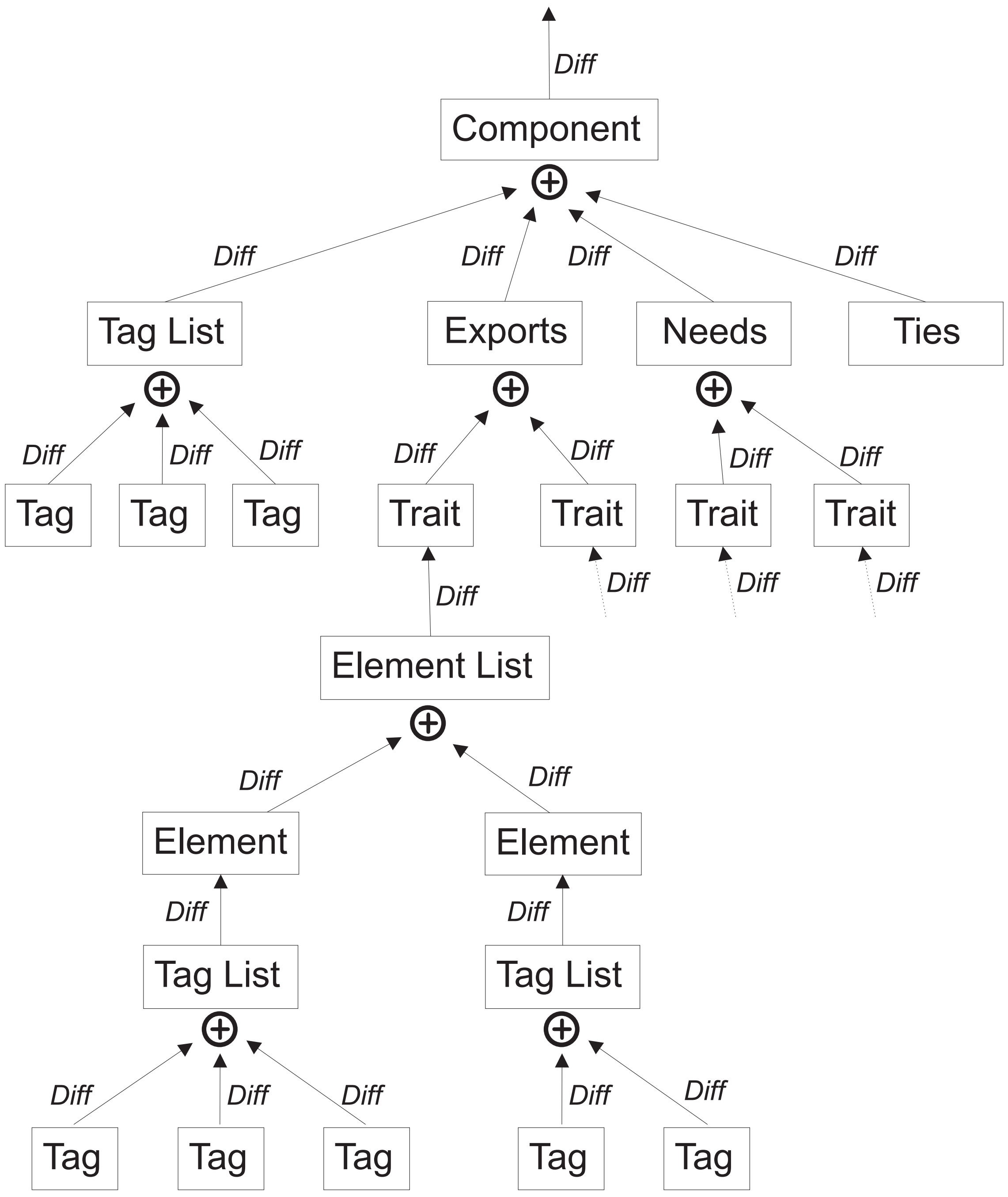

Figure 1: Schema of ENT comparison process

it can replace the original one. This method is used to ensure **strict** component substitutability but because of the complex output it can help to check whether the new component is replaceable in the particular **context** at least[3].

4.2.1 Subtyping rules

This general implementation of ENT comparison provides functions for subtype relation checking. It is needed to define and implement the check for various metatypes:

String Comparison of generic type String (any character string) is simple. Two strings can be either exactly the same, than there is *none* difference or else the difference is *mutation*.

Enumeration In this case the value of some interface part (tag, feature, ...) can gather just one value from the well-defined value set. The subtyping relation between values of that set must be specified. As an example we use the OSGi tag $resolution \in \{mandatory, optional\}$, where $mandatory <: optional$[4].

If the value does not change, difference is *none* indeed. If the value changes from A to B, consequent difference depends on the subtyping relation between these values (as defined in Section 3.1): $A <: B \rightarrow generalization$, $B <: A \rightarrow specialization$.

List We want to compare two lists of comparable items, e.g. list of exported Java interfaces or a list of supported operation systems. The complete algorithm of list comparison is quite extensive [17], we will introduce it simply at this place. Firstly we have to compare appropriate list items and combine gained differences. Than combine resulting difference with *insertion* if some new item was added to the list. On the contrary, if any of items was deleted, combine the difference with *deletion*.

It is important to understand the purpose of the concrete list. The items can be bound with each other in two diverse logical ways. Imagine the example list of imported Java interfaces: "Component requires Java interfaces `interface1` AND `interface2` AND `interface3`". Now imagine list

[3]e.g. when we know that the `getItemCount()` method is not used in the context of our application, we can proclaim that the new component can replace the old one in our application context

[4]This subtyping relation is bright at first sight. When import resolution changes from *mandatory* to *optional*, it means *generalization* of this interface element. In contrary case the difference is *specialization*

of required execution environments. By accordance with the OSGi specification, we must interpret the list this way: "Component requires execution environment `java.micro` OR `j2se` OR `PersonalJava`".

As shown in the examples, items can be bound with logical conjunction (AND) or disjunction (OR). The comparison algorithm must take this into account and behave different for logical disjunction.

Map As a map we understand a key-value pairs. In contrast to list, we always connect the items with conjunction. When the key is added/removed from the map, resulting difference is *insertion/deletion*. In the case of changed value, the *mutation* goes out.

Java class Java classes are being loaded from binary `.class` files and compared using introspection via Java Reflection API. The complete and complex comparison algorithm of Java classes and all its portions (methods, attributes, exceptions, ...) is described at [17].

4.3 ENT Representation of the Component

The package `metamodel` contains classes representing the component. The changes include the support for ENT-based component comparison described at the previous section. Classes thus implement the `ENTComparable` or `ENTComparableByName` interface and contain the generic comparison logic[5].

- `Component implements ENTComparableByName`
  Method `compareENT()` gets the differences of component level tags (from `TagSet`) and the traits (from `TraitsByENT`), combines them and returns the difference result of the whole component.

- `Element implements ENTComparableByName`
  The generic implementation gets the difference of element's tags (from `TagSet`) and returns it as a result of element's difference. This method must be overridden for a concrete element of a concrete component model.

- `Tag implements ENTComparableByName`
  The generic implementation is empty and it is also needed to override it in a concrete component model implementation.

- `ElementSet, TagSet implements ENTComparable`
  Both classes are sets of `ENTComparable` items, thus they are compared as the metatype **List** (see the Page 4.2.1).

[5]the concrete comparison algorithms must be derived for every component model separately

- `Trait implements ENTComparable`
  This is the new class. In the first version of implementation this class was not necessary. It represents trait of component interface, so it is a set of elements belonging to one trait definition. As it is explained in [16], this step between the component and its elements was omitted. For the purpose of ENT component comparison, we need to sort elements to their traits.

- `TraitSet implements ENTComparable`
  Collection of the traits, its behaviour is similar to `ElementSet` and `TagSet`.

- `TraitsByENT implements ENTComparable`
  Represents component traits sorted to the three categories according to their *role* in the component interaction - *exports*, *needs* and *ties*. This is important because comparison of traits depends on their role.

5 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 [10].

The first section describes implementation of the ENT metamodel for the OSGi component model. It is also an example of how to implement any component model's ENT representation and the related component comparison. There is also described the way of gathering information about existing OSGi component - generating ENT representation from its distribution form.

Second section focuses on the relationship between the differences of the component (or just only its part) and the versioning schema.

Next sections bound these two areas together - the method of the OSGi component comparison is described. As was already mentioned in the Section 2.3, the OSGi framework guarantee the safe component upgrades in the case of valid bundles. If the bundle's version identifier was assigned a wrong number, the upgrade/deployment process would finish sucessfully and the potential incompatiblity would cause the runtime error. Our method analyses the changes between given components and determines the subtyping relation between them. On the basis of this, the new version identifiers of the ”new” component (and all its versioned interfaces) are correctly assigned.

5.1 ENT Implementation for OSGi Component Model

In the mentioned master thesis [16] was implemented generation of Enterprise JavaBeans components ENT representation. Similarly, same functionality for component model OSGi Release 4 is presented now. Definition of this model in the terms of ENT can be found at Appendix A.

ENT metamodel definition for OSGi Release 4 is located in `osgi4` package, tools for retrieving information from JAR file and its OSGi manifest can be found in `osgi4.jar` package.

As mentioned in Section 4.3, subroutines for particular elements and tags comparison must be implemented for every component model separately. Classes which represents individual OSGi elements and tags are stored in packages `osgi4.elements` and `osgi4.tags`. They inherit from classes `metamodel.Element` and `metamodel.Tag` and introduce the specific implementation of `compareENT()` method. Its behaviour depends directly on the metatype of the element/tag. For example, `compareENT()` method of the `osgi4.tags.TagResolution` class have to compare enumeration metatype, which is defined by the subtype relation *mandatory* <: *optional*.

This part of OSGi module ensures the correct functionality of ENT based comparison of two OSGi bundles[6]. As a result of it we get the complete comparison result as the instance of interface `DiffResultPart`. It can be saved to XML file or be used as a source for additional operations. The simple example of the program usage follows:

```
/* generate the ENT representation of both components */
OSGiComponent component1 = getComponentRepresentation(jarFile1);
OSGiComponent component2 = getComponentRepresentation(jarFile2);

/* compare components */
DiffResultPart diffResult;
diffResult = component1.compareENT(component2);

/* save result to XML file */
diffResult.saveToXML(outputXMLFile);
```

The second functionality of this module is loading the ENT representation from the distribution form of the OSGi bundle. The ENT component representation - elements 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 input of this part is thus a JAR file which contains an OSGi bundle. It loads all information that is needed (and available) to build the ENT representation of

[6]their ENT representations

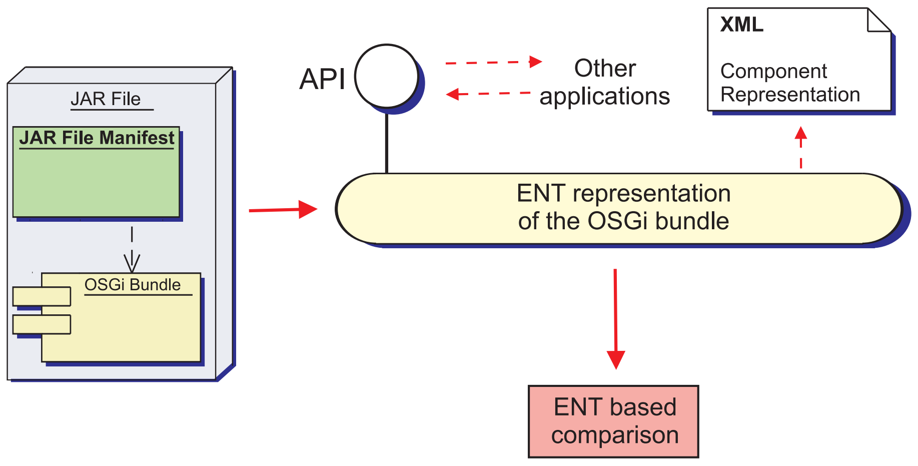

Figure 2: Comparison of the ENT representations

the bundle. The ENT definition of the OSGi component model (see Appendix A) is used, it contains (among others) the concrete hints for locating the required information in the JAR file. When the ENT representation is successfully created, it can be saved to XML file[7], displayed in the visualization tool or compared to the ENT representation of another OSGi bundle (see the Figure 2).

5.2 Differences and Versioning Related

The OSGi uses a well-known and widely used versioning schema *major.minor.micro* (see Section 2.3). The knowledge of difference between two subsequent versions of an interface part is sufficient to determine the new version identifier of that part. Let $d = Diff(R_{i-1}, R_i)$ be the difference between two consecutive revisions, $V_{old} = maj_{old}.min_{old}.mic_{old}$. Than the new version identifier $V_{new} = maj_{new}.min_{new}.mic_{new}$ is defined by the rules in Table 3.

$\mathbf{Diff(R_{i-1}, R_i)}$	maj_{new}	min_{new}	mic_{new}
none	maj_{old}	min_{old}	$mic_{old} + 1$
specialization, insertion	maj_{old}	$min_{old} + 1$	0
deletion, generalization, mutation	$maj_{old} + 1$	0	0
unknown	unknown		

Table 3: Derivation of the new version identifier

For example, assume that we have the package `cz.zcu.logging` and that version 1 of the package has version identifier 1.2.1. If the content of package changes

[7]the ".emr" format specified in [3]

to its version 2 for a new release, the *mutation* difference between both versions signals an incompatible change. Therefore, the new release of package `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 *specialization* change in the package content, 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.

5.3 Practical Use Case

Previous chapters described the implementation of ENT component comparison. The result of that process can be used in many ways - one of it will be presented now. As it was already mentioned, OSGi bundles use the versioned dependencies - the version identifier is assigned to the whole bundle as well as to its exported interfaces. After the new component version development, it is necesary to set the new version identifiers of all these versioned objects. This process can be easily automated because new version identifier is a function of the old one and the difference between component versions (see Section 5.2 on the Page 21).

The input of the whole process are two subsequent releases of OSGi bundle - two JAR files. The implementation of the ENT comparison (Section 4.2) and the implementation of the OSGi component model (Section 5.1) will be used to load and compare those two components. The result is stored in the instance of `DiffResultPart` class. This object contains the differences of all elements, traits, categories and the component itself. On the basis of these differences and the old version identifiers we have to determine the new version identifiers.

5.4 Algorithm of Comparison

In this section we present the algorithm for assigning the new version identifiers of an OSGi bundle. 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 (see Figure 3 for schema):

1. Load the bundles and create their ENT representations.
2. Compare those representations using the program described in Section 4.2 and save the difference result (`DiffResultPart` instance).

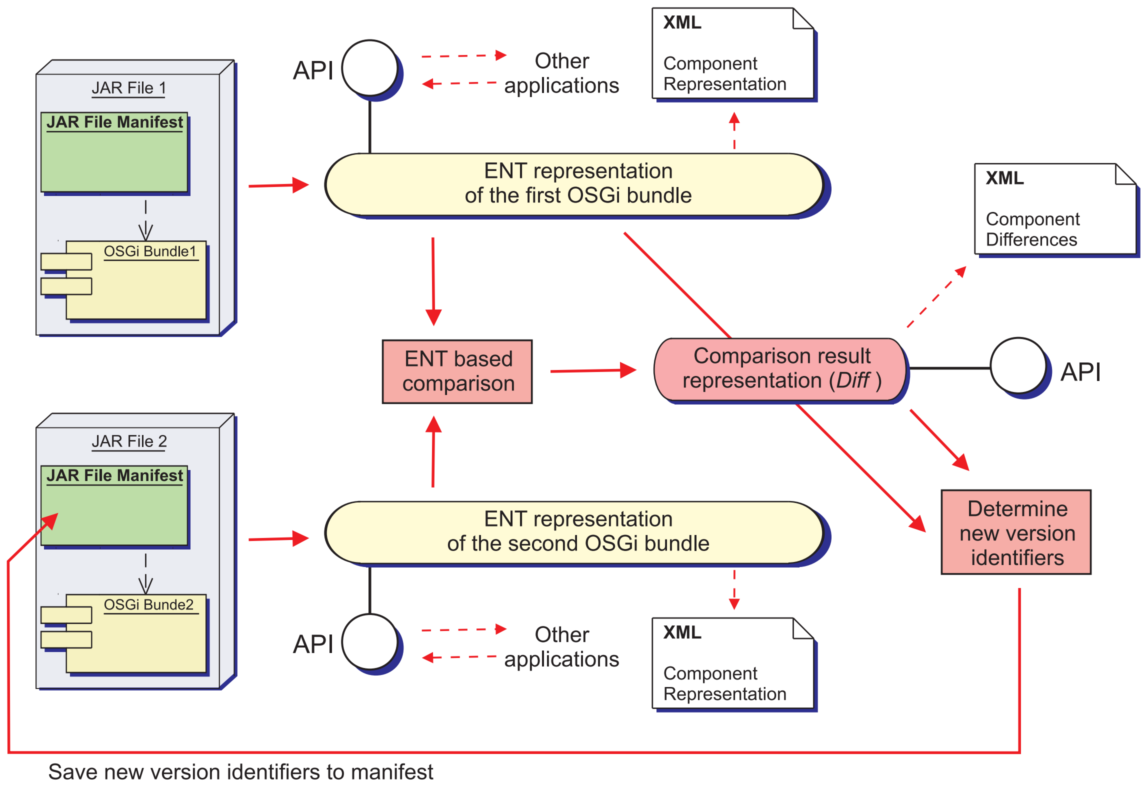

Figure 3: Schema of the comparison algorithm

3. Generate the new version identifier of the whole component A_2 according to Table 3. The difference of the whole component take from the root level of the `DiffResultPart` structure.

4. Traverse the `DiffResultPart` structure and find the differences of *export types* trait and combine those referring to the same Java packages. As a result of this you gain the difference of every package exported by component.

5. Generate the new version identifiers of each of those packages according to Table 3.

5.5 Example of the OSGi Component Comparison

Imagine two versions of `LogService` bundle. Fragments of the manifest files follow:

- Old version:

```
Bundle-Name: LogService
Bundle-Version: 2.3.2
Export-Package:
  cz.zcu.logging;version="1.3.0"
Import-Package:
  org.osgi.framework;version="[1.3, 2)"
```

- New version:

```
Bundle-Name: LogService
Bundle-Version: ?.?.?                <to be determined>
Export-Package:
  cz.zcu.logging;version="?.?.?"    <to be determined>
Import-Package:
  org.osgi.framework;version="[1.3, 2)"
Require-Bundle: new_bundle_dependency
```

We have to determine the version identifiers denoted by the note `<to be determined>`. Suppose the changes in the `cz.zcu.logging` presented at Figure 4. The difference of this package is *Specialization* and its new verson identifier is `1.4.0`.

Tables 4 and 5 show the traits and tags. Every trait has either provided (P) or required (R) role. Tables remind all the features and non-functional tags that have to be taken in account.

Additional tags are attached to the component itself as well as to its particular features. Tags have to be compared using metatypes defined in the table 5.

Detailed specification of OSGi interface traits and tags is located at Appendix A. There is only one change in the example - new dependency to an OSGi bundle

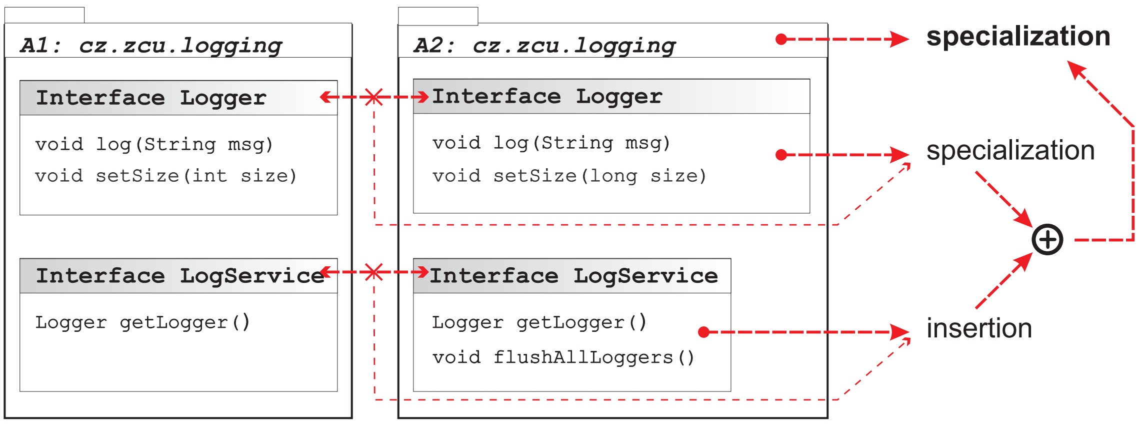

Figure 4: Example of package `cz.zcu.logging` comparison

Feature	Role	Metatype compared
export types	P	Java class, tags
import types	R	Java class, tags
native code	R	List, tags
require bundles	R	String (bundle name), tags
required exec env	R	List

Table 4: Summary of the OSGi traits

(`Require_Bundle` manifest header). Because of it, the difference of the whole component is *mutation* and its the new version is `3.0.0`.

5.5.1 Subtyping Rules for OSGi Specific Metatypes

The *Type compared* column describes what is compared - Java type, String, Map or List of Strings or other aspect of component interface. The subtyping rules for common metatypes were defined at Section 4.2.1, at this place we present the rules for the OSGi-specific metatypes:

Version interval The same intervals generates the *none* difference indeed. If the version interval in the second component includes the first component's interval (in other words it has been extended), difference is *generalization*. In the contrary situation (inteval has been narrowed), it is *specialization*. In any other case, resulting difference is *mutation*.

Import types In contrast to export types we do not have access to type implementation. The only information available is the type name and (optionally) required version interval. We therefore compare import types as a list of strings with regard to versions, if present.

Tag	**Belongs to trait**	**Metatype compared**
bundle symbolic name	import type	String
bundle version	import type, require bundle	Version interval
language	native code	List
os name	native code	List
os version	native code	List
parameters	export type, import type	Map
processor	native code	List
resolution	import type, require bundle	Enumeration
symbolic name	bundle	String
version range	import type	Version interval
kind	import type	Enumeration

Table 5: Summary of the OSGi tags

5.6 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.

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).

6 Related Work

The area of component substitutability checking is well researched.

Zenger [20] 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 [7]. 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 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 [2]. 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 [18]. 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 [8]. 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.

7 Conclusion

The implementation of ENT metamodel and ENT based comparison allows the substitutability checking of component at the metamodel level. One can define traits and tags for any other component model and implement particular subtype checking methods - and the new component model would be supported.

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.

A OSGi Platform Release 4: the ENT meta-model

This chapter contains the specification of the OSGi Service Platform Release 4 [10] in the terms of the ENT meta-model. It consists of the definition of their trait set and of tags on the component level.

A single component in the OSGi Service Platform is denoted by the word *bundle*. Bundles are distributed in the form of JAR file with defined structure. All information about the bundle are stored in the default JAR file manifest.

OSGi Specification Release 4 published in August 2006 presents many innovations and enhancements, but it is still backward compatible with all older versions. Emphasis has been put especially on bundle versioning, security (support for signed bundles, new permissions), localization and new bundle types (fragment and extension bundles).

In our point of view the most important novelty is the detailed specification of versioning. Not only the whole bundle has its version identifier, but bundle's particular exported packages can also have it. System of framework classloaders allows use of more versions of one Java package when different versions are required by bundles. Bundle can specify many constraints for exported and imported packages - required package and/or bundle version, package dependencies, mandatory and optional imports and so on. These things help the framework to select the best connections between bundles at runtime and are also useful as the definition of bundle's interface.

Specification defines one component type called *Bundle* (for now we didn't take into account two new auxiliary bundle types - fragment and extension bundles. They will be studied later).

A.1 Grammars used

Version and version range specifications are used in several places of the following text. We will define the grammar of those tokens:

version. Version token has the following grammar (as defined in [10] on page 30):

```
version   ::= major( '.' minor ( '.' micro ( '.' qualifier )? )? )?
major     ::= number
minor     ::= number
micro     ::= number
qualifier ::= ( alphanum | _ | '-' )+
```

Default value is "0.0.0".

version-range. A version range describes a range of versions using a mathematical interval notation. (as defined in [10] on page 30):

```
version-range::= interval | atleast
interval::= ( '[' | '(' ) floor ',' ceiling ( ']' | ')' )
atleast::= version
floor::= version
ceiling::= version
```

Default value is "[0.0.0, inf)".

A.2 Tag definitions

In contrast to previous version of OSGi specification, we can find some important information about the whole bundle now:

symbolic_name metatype = *string*

Symbolic name specifies unique, non-localizable name for this bundle. This name should be based on the reverse domain name convention, see [10] on page 37. There is no default value for this tag because this information about bundle is mandatory (correct bundle must specify its symbolic name).

version. metatype = $versionidentifier$

Version of the bundle. Source: *Bundle-Version* manifest header.

A.3 Trait definitions

Trait definitions are ordered alphabetically by the trait name.

export_types – trait specifies Java types (classes or interfaces) to be exposed to other bundles.

metatype = *class*
classifier = ($\{syntax\}$, $\{operational\}$, $\{provided\}$, $\{structure\}$, $\{type\}$, $\{permanent\}$, $\{multiple\}$, $Lifecycle$)
tags: version = $version$, parameters = $string$

Notes:

1. Source: Manifest header *Export-Package* (see the specification page 39) and read notes bellow.

Tags:

- *version*: The version of the type with syntax as defined on page 28. The default value is 0.0.0.
  metatype = $versionidentifier$
- *parameters*: contains mandatory parameters of the package export as a map using format `( attribute=value ',' )*`. This means those specified as mandatory by the attribute *mandatory* (see specification at page 47)
  metatype = *map*

import_types – importing allows a bundle to request access to types that have been exported by other bundles.

metatype = *class*
classifier = $(\{syntax\}, \{operational\}, \{required\}, \{structure\}, \{type\}, \{permanent\}, \{single\}, Lifecycle)$
tags: bundle_symbolic_name = *string*, bundle_version = $versionrange$, kind $\in$ $\{static, dynamic\}$, parameters = *string*, resolution $\in$ $\{mandatory, optional\}$ (default value = *mandatory*), version_range = $versionrange$

Notes:

1. Source: Manifest header *Import-Package* (see the specification page 38) or *DynamicImport-Package* (page 53) and read notes bellow.
2. Tag *parameters* contains parameters of the package import as a map using format `( attribute=value ',' )*`.

Tags:

- *bundle_symbolic_name*: symbolic name of required exporter.
  metatype = *string*
- *bundle_version*: version range of required exporter.
  metatype = $versioninterval$
- *kind*: OSGi allows two types of imports - static and dynamic. This tag differentiate between them.
  metatype = $enumeration$
- *parameters*: contains parameters of the package import as a map using format `( attribute=value ',' )*`.
  metatype = *map*
- *resolution*: import statement can be mandatory or optional. So this tag reflects the *resolution* directive of *Import-Package* manifest header.
  metatype = $enumeration$

- *version_range*: a version-range to select the exporter's package version. The syntax must follow *Version Ranges* on page 29. For more information on version selection, see *Version Matching* on page 42 of the OSGi specification. Source of this tag is the *version* import directive of *Import-Package* manifest header.

  metatype = $versioninterval$

native_code – elements of this trait describe references to native code libraries which are requested by the bundle.

metatype = *string*
classifier = $(\{\mathit{syntax}\}, \{\mathit{operational}\}, \{\mathit{required}\}, \{\mathit{item}\}, \{\mathit{instance}\}, \{\mathit{permanent}\}, \{\mathit{single}\}, \{\mathit{development}, \mathit{assembly}, \mathit{deployment}, \mathit{runtime}\})$
tags: language = *string*, processor = *string*, osname = *string*, osversion = *string*, selection_filter = *string*

Notes:

1. Source: Manifest header *Bundle-NativeCode*, see the specification page 59.
2. All the additional information provided by the bundle developer are stored as tags. Type of the element is name of the referenced native code library (as specified in the manifest header).

Tags:

- *language, processor, osname, osversion*: all these tags specify the required native code library - its required localization language, processor family and operation system. Tag value is the comma separated list of possible values (connected by disjunction). For example: *osname="win95,win98,winxp"*, which means that required library is intended to work at one of specified operation systems. Empty string or unspecified tag value signals that the particular condition is not used.

  metatype = *list*
- *selection_filter*: more complex selection filter of the native code library. See the specification on the page 60.

  metatype = *string*

require_bundles – bundle should require all types exported by another bundle.

metatype = *string*
classifier = $(\{\mathit{syntax}\}, \{\mathit{operational}\}, \{\mathit{required}\}, \{\mathit{structure}\}, \{\mathit{instance}\}, \{\mathit{permanent}\}, \{\mathit{single}\}, Lifecycle)$
tags: resolution $\in \{\mathit{mandatory}, \mathit{optional}\}$ (default value = *mandatory*), bundle_version = $versionrange$

Notes:

1. Source: Manifest header *Require-Bundle*, see the specification page 66.
2. See notes below

Tags:

- *bundle_version*: version range of required bundle.
  metatype = $versioninterval$
- *resolution*: *Require bundle* statement can be mandatory or optional. So this tag reflects the *resolution* directive of *Require-Bundle* manifest header.
  metatype = $enumeration$

required_execution_environments – bundles can be restricted to one of specified execution environments.

metatype = *set*
classifier = $(\{syntax\}, \{operational\}, \{required\}, \{item\}, \{type\}, \{permanent\}, \{single\}, \{development, assembly, deployment, runtime\})$
tags = $\emptyset$

Notes:

1. Source: Manifest header *Bundle-RequiredExecutionEnvironment*, see the specification page 33.
2. The plaform assumes that if the information about the required execution environment is not present in the bundle manifest, the bundle can be deployed to any environment which conforms to the standard OSGi release 4 platform specification. If multiple values are specified, the deployment environment name must match at least one of the values (i.e. logical OR is performed, not an AND operation).

use_packages – this trait captures package dependencies between bundles.

metatype = *map*
classifier = $(\{syntax\}, \{operational\}, \{provided, required\}, \{item\}, \{instance\}, \{permanent\}, \{single\}, Lifecycle)$
tags = $\emptyset$

Notes:

1. Source: Manifest header *Export-Package*, attribute *uses*, see the specification page 39.
2. Element types of this trait are in the form of map *packageA:packageB* which means *packageA* requires *packageB*.

A.4 Notes on the OSGi Service Platform Release 4

The meta-types used in the specification have the following meaning. The *class* correspond to the Java classes or interfaces, the *string* denotes any string (e.g. name of the referenced library). Finally, *map* is to be interpreted as "set of name-value pairs".

In comparison for example with Enterprise JavaBeans component model [13], OSGi Platform is much simpler. Because of this, its ENT meta-model is not complicated, it contains only six traits.

There is one feature of OSGi that requires several additional notes - dynamic package importing. Bundles must use the static import (*import_package* trait) when the names of referenced packages are known before bundle has started. If this information is known only at run-time, bundles should define set of possible dynamic imports in the *DynamicImport-Package* manifest header. In contrast to static import, references to required packages can use asterisk notation. Bundle can therefore define, that it wants to dynamic import package `com.httpserver.*` (this means "every package name beginning with prefix `com.httpserver.`"), `com.*` or even `*`. Meaning of this definition is simple but useless - bundle can import every package at run-time. Despite this we decided to model this feature in the ENT meta-model view, although it does not carry any valuable information. Even this is the part of component interface and thus it is modelled.

One of the most important innovations in OSGi Release 4, which is also closely connected with ENT modelling, is the *require_bundle* trait. Beyond the "classic" and recommended way of importing Java packages (*import_types* trait) this allows bundle developer to import all types (Java classes and interfaces) exported by selected bundle. Although specification does not recommend it and shows many issues with requiring bundles (see specification chapter 3.13.2), it must be considered when creating the ENT model of OSGi Release 4.

The required bundle is determined by its symbolic name and by a few parameters (not important from our point of view). Because the binding between exporter and importer bundle is created at the resolve phase of deployment, we cannot know concrete types which are imported and used by client bundle before it is deployed and started in the framework environment. Symbolic name of the bundle doesn't have any relation to the types being exported in general. As a conclusion we have to say that from the symbolic name of required bundle we are not able to gain information about the concrete types – classes and interfaces which will be imported from the referenced bundle and used at runtime. We must therefore model only the symbolic name of required bundle as the type of interface element of the trait *require_bundle*.

A.5 How to discover import and export types

In the manifest file of each bundle there are three relevant headers: *Import-Package*, *DynamicImport-Package* and *Export-Package.* They specify set of package names to be imported and exported. We decided that this information is not so useful because package name does not represent any real type, but a collection of "some" classes and interfaces.

When analyzing a bundle in order to build its ENT representation, it is necessary to discover concrete classes and interfaces "hidden" behind these package names (which are too general).

A tool or human trying to find all classes and interfaces for *export* must follow this process:

1. For every package name specified in bundle's *Export-Package* manifest header:
    (a) Browse bundle's JAR file and find the directory of this java package. For example, package `org.osgi.test` is in the directory `JAR://org/osgi/test/` by default. But another manifest header must be also taken in account: *Bundle-ClassPath.* It specifies custom search paths in bundle's JAR file.
    (b) Find and save as a new *export_types* element every public class or interface found in that directory.

Identifying classes and interfaces for static and dynamic *import* is rather complicated and not so reliable, but in most cases it works fine. The idea of this algorithm is simple: We must discover all classes and interfaces which are used by this bundle and filter the ones from packages specified in *Import-Package* or *DynamicImport-Package* manifest header. The begin of the search is bundle's "main" class called activator - its name is specified by the *Bundle-Activator* manifest header.

1. Let *List* be a set of class names, which have not yet been parsed.
   Let $Parsed$ be a set of class names already parsed.
2. $List = \{\text{Bundle-Activator Class}\}, Parsed = \emptyset$
3. $Class$ = one class name from *List*, move this item from *List* to $Parsed$
    (a) If class $Class$ belongs to this bundle, parse it. Find all references to another classes or interfaces and add them to *List* (only if they are not already in $Parsed$ list).

(b) If class *Class* belongs to one of packages specified by *Import-Package*, save it as a new *import_types* element with tag *kind* = *static* and continue.

(c) If class *Class* belongs to one of packages specified by *DynamicImport-Package*, save it as a new *import_types* element with tag *kind* = *dynamic* and continue.

(d) Ignore class *Class* otherwise and continue.

4. Finish process if *List* is empty, jump to step 3 otherwise.

To implement this procedure it is necessary to parse java byte-code files (those with `.class` extension). For this purpose, one of many open source projects can be used. In our prototype implementation we used The Byte Code Engineering Library [1].

B XML Representation

This section contains document type definitions (DTDs) for the XML representation of ENT structures.

B.1 ENT Differences

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- root element, which represents differences between two components -->
<!ELEMENT entdiff (component)*>
<!-- this element represents the compared component -->
<!ELEMENT component (provider, namespace, name, description?, part+)>
<!-- component model and component type -->
<!ATTLIST component
    model CDATA #REQUIRED
    ctype CDATA #REQUIRED>
<!-- provider of the component -->
<!ELEMENT provider (#PCDATA)>
<!-- namespace of the component -->
<!ELEMENT namespace (#PCDATA)>
<!-- name of the component -->
<!ELEMENT name (#PCDATA)>
<!-- description of the component -->
<!ELEMENT description (#PCDATA)>
<!--
    This element is used at many levels:
    "component" - level of the whole component
    "categories" - here are stored the differences between
    all categories sets (Exports, Needs and Ties)
    of the compared components
    "tags" - difference between all tags of components
-->
<!ELEMENT part (diff,info?,part*)>
<!-- name of part -->
<!ATTLIST part
    name CDATA #REQUIRED
    level CDATA #REQUIRED>
<!-- Optional textual information -->
<!ELEMENT info (#PCDATA)>
<!--
Difference between two objects (tags, traits, categories, components, ...)
must be one of this:
    "None"
    "Insertion"
    "Specialization"
    "Deletion"
    "Generalization"
    "Mutation"
    "Unknown"
-->
<!ELEMENT diff (#PCDATA)>
```

References

[1] The Byte Code Engineering Library,
http://jakarta.apache.org/bcel/.

[2] Premysl Brada. *Specification-Based Component Substitutability and Revision Identification*, PhD thesis, Department of Computer Science, University of Western Bohemia, Pilsen, Czech Republic, August 2003
Available at http://www.kiv.zcu.cz/˜brada/research/thesis/.

[3] P. Brada. *The ENT model: A general model for software interface structuring.* Technical Report DCSE/TR-2002-10, Department of Computer Science and Engineering, University of West Bohemia, Pilsen, Czech Republic, 2002.

[4] Premysl Brada. Issues in Static Verification of Component Substitutability. In *Proceedings of Objekty 2005*, Ostrava, Czech Republic, November 2005.

[5] *Enterprise Component Framework*, Available at http://www.eclipse.org/proposals/ecp/.

[6] L. Cardelli. *Type Systems*, Handbook of Computer Science and Engineering, Chapter 103. CRC Press, 1997.

[7] S. McCamant, 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, USA, 2004.

[8] The Open Group, *Distributed Computing Environment (DCE) 1.1: Remote Procedure Call*, Available at http://www.opengroup.org/dce/.

[9] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 3.* March 2003,
Available at http://www.osgi.org/.

[10] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.* August 2005,
Available at http://www.osgi.org/.

[11] *Eclipse Platform*,
Available at http://www.eclipse.org/platform/.

[12] *Equinox project.* Eclipse Platform,
Available at http://www.eclipse.org/equinox/.

[13] Sun Microsystems. *Enterprise JavaBeans^{TM} Specification, Version 2.1*, listopad 2003
Available at http://java.sun.com/products/ejb/.

[14] Sun Microsystems. *Java Platform, Micro Edition*
Available at http://java.sun.com/j2me/.

[15] Object Management Group, *CORBA Component Model, V3.0*,
Available at http://www.omg.org/technology/documents/formal/components.htm.

[16] Lukas Valenta, *Modeling of EJB Components*, Master thesis, Department of Computer Science, University of Western Bohemia, Pilsen, Czech Republic, July 2005.

[17] Pavel Stuna, *Verification of EJB components substitutability*, Master thesis, Department of Computer Science, University of Western Bohemia, Pilsen, Czech Republic, July 2005.

[18] Sun Microsystems, *Package Version Identification*, available at http://java.sun.com/.

[19] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 5577. Springer-Verlag, 1988.

[20] M. Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.