Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 264 (4) (2011) 3–18
www.elsevier.com/locate/entcs

# Reconstruction of Type Information from Java Bytecode for Component Compatibility [1]

Jaroslav Bauml, Premek Brada

{jbauml|brada}@kiv.zcu.cz
*Department of Computer Science and Engineering*
*University of West Bohemia*
*Univerzitni 8, 30614, Pilsen, Czech Republic*

**Abstract**

The Java type system is strictly checked by both the compiler and the runtime bytecode interpreter of the JVM. These mechanisms together guarantee appropriate usage of class instances. Using modern component systems can however circumvent these static checks, because incompatible versions of classes can be bound together during component installation or update. Such problematic bindings result in ClassCastException or NoSuchMethodException runtime errors. In this paper we describe a representation of Java language types suitable for checking component compatibility. The presented approach applies various bytecode handling techniques to reconstruct a representation of the Java types contained in a component implementation, using different sources of class data. The representation is then used during build- and run-time type system verifications with the aim to prevent these kinds of errors. We have successfully applied this approach to prevent OSGi component incompatibilities.

*Keywords:* type reconstruction, reflection, bytecode analysis, subtyping, component compatibility

## 1 Introduction

Statically typed languages have clear advantages for which they are used in the majority of software systems. As Erik Allen describes clearly in [1], static type checking improves robustness through early error detection, increases performance by making the required checks at the best time and supplements the weaknesses of unit testing. Early checks of type coherence done by compiler ensure type safety of the program code and guarantee that types used at runtime are compatible.

This clear situation is however complicated by component systems. One of the most important contributions of Component-Based Software Engineering (CBSE) [20,4] is the decomposition of applications into smaller parts – components. An application is not built and deployed as one monolithic block but composed from

[1] This work was supported by the Grant Agency of the Czech Republic under grant 201/08/0266.


doi:10.1016/j.entcs.2011.02.002

components which encapsulate parts of its functionality, possibly developed by independent vendors. Each component has its own interface which is split into two sides – the sets of provided and required features. Through these features, components are wired together according to their declared dependencies. Today more and more Java based systems move to this kind of modularized or component-based architecture, supported by systems like OSGi, Netbeans plugins or Android application architecture.

At component deployment time, a problem stemming from type mismatches can occur in case the structure of a type exported by a providing component changes during its evolution. The client components will still be wired to such provider (since the type names in the provided-required feature pairs match) but the provided language type can now be incompatible with the notion of this (referenced) type on the client side.

As we show in [8] and [5], this scenario is realistic in case of independent component evolution. We have therefore proposed a method for deciding on component compatibility based on the subtyping comparison of the real structure of the referenced type and the described client's notion of that type. Such run-time compatibility checks depend on the complete reconstruction of type description from component binary implementation – during and after deployment, its source code is rarely accessible.

In this paper, we discuss in detail the alternative ways that exist for the reconstruction of Java types by bytecode analysis and run-time introspection. The following Section 2 focuses deeply on the motivating problem with real life examples. The features which are utilized in component dependencies actually depend on the component model used. Since we work mainly with the OSGi component model, we will provide short description of OSGi in subsection 2.3.

The proposed method of Java type reconstruction is described in the next two sections. Section 3 describes a Java type system representation employed by our method. The generality of its design allows the representation to be used in other projects to ensure Java language type compatibility. Section 4 describes the approach to deciding on component compatibility which uses algorithms working on the type system representation. The merits of these methods are discussed in the end of the work.

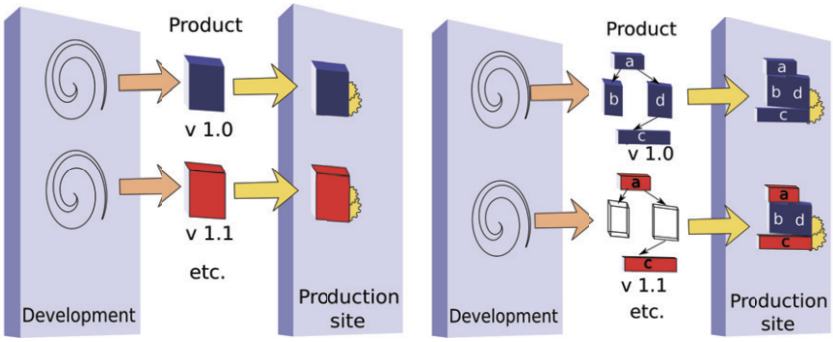
## 2 Compatibility in Component Software

In the industry and research worlds there are various component models which differ from each other by complexity, level of abstractness, technical maturity and the purpose of use. But in each of these different component models one implicit requirement is shared – it is component compatibility.

### *2.1 Component Dependencies*

Component compatibility is a crucial requirement because of component life cycle. As shown in Figure 1a, when working with standard monolithic software the depen-

dencies between its parts (let say classes) are created at build time when they are also checked by the compiler to be type compatible. In case of problems found by the compiler, the resulting code is not created; when language types (classes and interfaces) are compatible, the application can be built and deployed to a production site. When a new version of software is developed, the whole monolithic code package is again moved to the production site for application upgrade.




(a) Monolithic software (b) Component software

Fig. 1. Software process

When using component software development techniques the situation is similar in general but differs in important details. Each component has its own interface which is composed of two sides – the provided and required one. The provided side consists of features (services, packages, events, ...) which are published by the component so as to be available to component surroundings. Conversely, the required side declares features which the surrounding must suply to the component for its proper function. Through these features, components are wired together, forming dependencies in a way described by the particular component model.

### *2.2 Type Compatibility in Component Applications*

As can be seen in Figure 1b the dependencies between components are also at first checked by compiler at component build time. Unlike the monolithic application scenario however, each component (including those developed by third-party vendors) can then be deployed to the production site separately – without its depended-on suppliers, or to be precise, only with a declaration of these dependencies.

During subsequent update of the component-based application, the wirings among components are re-established at the production site with the component in the new version. When the new version is incompatible, the application will fail with some kind of runtime exception. The probability of the failure is equal to the probability of invoking the type (e.g. class) which exhibits the incompatibility.

In many currently used component models, there is no mechanism to describe the type system of component interface. The types from the provided part of a supplier component are used in a client's source code, creating an implicit "notion" of these types in client's implementation. This binding of the client's code to the supplier's types is logical but creates an invisible static dependency in the client

implementation – its notion of the referenced types is based on the structure of the particular supplier's types used during compilation. At compile time, the actual types exported by the supplier and this notion are checked for coherence. However, an analogous mechanism is missing during the component (re)wiring operation in the deployment phase.

This general problem is shared by many Java based component systems. From now on we will present it on a case study of the OSGi platform which is very simple and lightweight and its popularity is growing. For insight to the problems handled further, an elementary knowledge of OSGi is required; if you are familiar with the framework you can skip the next subsection.

### *2.3 A Brief Overview of OSGi*

The Open Services Gateway Initiative (OSGi) platform [18] is an open Java-based framework for service deployment and management. Its uses range from embedded applications to large-scale desktop and enterprise systems. The core of OSGi is the *framework* which creates a runtime environment for managing the deployment and lifecycle of components called *bundles*. A set of standardized basic services, implemented by system bundles, is provided as part of the framework distribution.

A bundle can export (provide) or import (require) Java packages and services, declare native libraries used, and specify dependencies on the execution platform and concrete bundles. The standard Java manifest file holds the specification metadata. *Packages* are used to access shared types and bundle implementation, and form static bindings between bundles. *Services* are represented by Java interfaces and allow dynamic registration, lookup and (un)binding of functionality using a centralised framework registry.

The onus of service binding is by default on the bundle implementation which brings flexibility in handling runtime changes. If a standardized declarative services module is used, service declaration and binding can be delegated to the framework. On the other hand, dependency resolution for packages is always handled by the framework core, requiring no work on the programmer's side.

### *2.4 Real Word Problem Example*

In this section we describe a concrete example of the problem with component compatibility, showing how the user can be affected by this issue. It is one instance of a set of runtime failures which take hours to track down. Methods which prevent such runtime exceptions can therefore save valuable amounts of development time.

To develop a frontend for a research project we decided to extend the Apache Felix Webconsole [2] bundle. This Webconsole is an extendable webpage for managing a running OSGi framework. It embeds a servlet container which can be extended by registering a service implementing an interface `org.apache.felix.webconsole.AbstractWebConsolePlugin`.

Our plugin bundle, called subst-verifier, is very simple. It can verify if a new version of a bundle is compatible with an old version. It has only one HTML form


Fig. 2. Real world compatibility problem

with file input. When the file is uploaded to the plugin, the verification is performed.

When we installed subst-verifier and tried to upload a file we got the error message shown in Figure 2. After several hours of problem searching we found the following issue to be its cause: We have used the *org.apache.felix.webconsole* bundle in version 1.2.10 which expects library `commons-fileupload` in version 1.1. This dependency is not handled by metadata description of the component and therefore it was not easy to observe it. Our subs-verifier was however compiled to `commons-fileupload` in version 1.2. Because these two versions of a widely used library are not compatible and no compatibility checks are made in OSGi component model, we got a serious runtime crash of our application.

## 3 Component Type-Level Representation

In order to compare two components and determine the level of their compatibility at runtime (when source files are not accessible), we need a suitable model to represent both components. The representation we describe in this section was designed to capture all syntactic changes of types on the public API of a component. It consists of two layers (see Figure 3). For the layer of the whole component we use a simple metamodel of OSGi called *BundleTypes*. It represents the exported and imported features of a bundle. The second layer describes only the Java types declared and used by the component – it is therefore called *JavaTypes*. Since all OSGi bundle features are implemented as or consist of Java classes, the leaf nodes of the BundleType layer use the Java type representation described by JavaTypes.

The *JavaTypes* layer can be used independently of the Bundle representation. We have designed this layer according to the Java Language Specification, Third Edition [14]. The JavaTypes layer is very similar to the Java reflection API [13] but is more general because the contents can be obtained from other sources than just reflection. For reasons described below, the unique feature of *JavaTypes* is the ability to create the type representation also from component bytecode or the possibility to create representation of nonexisting classes by manipulation.

There are some quite subtle differences between the Java language type system and the type system which JVM uses when interpreting bytecode. Since our method

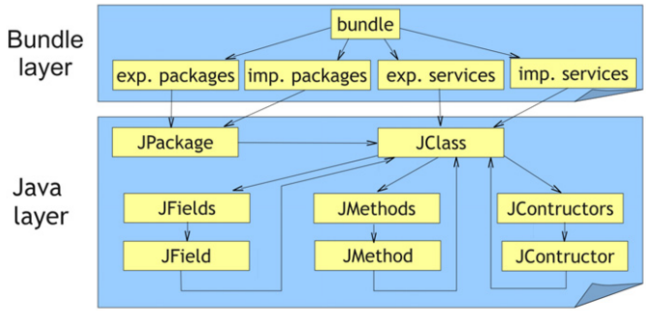



Fig. 3. Component Representation

is focused on reconstructing Java representation and reasoning over Java programing elements, for the rest of the paper we will use the Java type system.

The base interfaces of *JavaTypes* are shown in Figure 4. *JType* is the parent interface of specific types in Java. These specific types are: *JClass*, *JTypeVariable*, *JParametrizedType*, *JWildCardType* and *JGenericArrayType*. *JClass* represents basic types of Java language – a class or an interface. Other children of *JType* represent generic types. In the rest of the paper we will call types represented by *JClass* as basic types and other types as generic types.

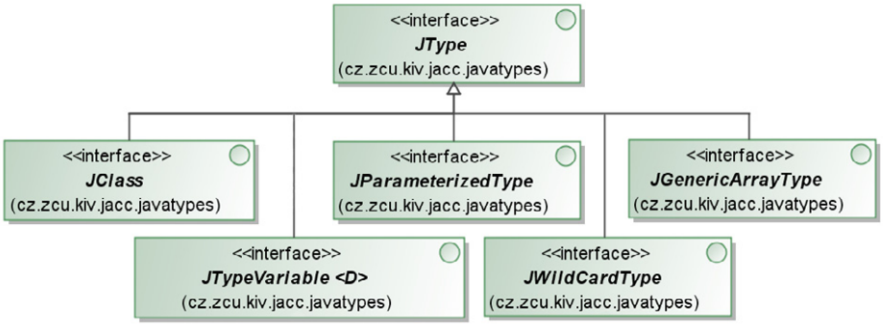



Fig. 4. Base interfaces of JavaTypes representation

To cover generic types, *JTypeVariable* represents a type variable in a class or interface definition, *JParametrizedType* covers the case of an instance with type variable in its definition, and *JWildCardType* represents an instance of type with a wildcard – eg. `List<?>`. Lastly, *JGenericArrayType* represents an array of *JTypeVariable*, *JParametrizedType* or *JWildCardType*. (The situation concerning generics is actually more complicated because there are cases when their representation is not available, e.g. through reconstruction from bytecode which does not contain generics annotations.)

The representation of language features available in Java is summarised in Figure 5. *JClasses* aggregate *JMembers* (*JFields*, *JMethods* or *JConstructors*). For all these elements an *JModifier* can be obtained, expressing their access modifier –

*public*, *private*, *static*, etc. All elements which can have an annotation attached have to implement the *JAnnotable* interface which can return an *JAnnotation* object.

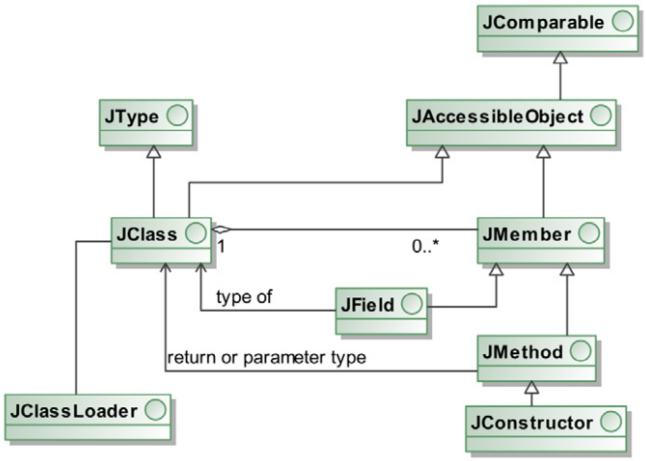



Fig. 5. Diagram of core JavaTypes representation layer

### *3.1 Type Representation Sources*

Creating the representation of types referenced by a component's interface is designed in a way similar to the Java class loading process. The types can be obtained from several sources. Each source has its *JTypeLoader* object that is responsible for reading the types located in the given source. The currently implemented sources for retrieving *JavaTypes* representation are:

- A loaded program – through Java reflection API;
- Compiled but not loaded files – using bytecode inspection;
- Programmatically – custom creation, for cases like testing or stub creation.

Furthermore, each JType loader has its parent *JTypeLoader*. The tree organization of loaders and the fact that each loader can create the representation from a different source brings variability into the process of obtaining the component's type representation. For example, when a referenced class is not found inside the component's bytecode we can try to find it through the parent JType loader. This can be a reflection loader which creates the class representation in cooperation with a classloader pointed to a classpath (where the class is available). These sources can be arbitrarily combined together.

Retrieving representation using *Java reflection* is quite straightforward because of the intentional similarity between JavaTypes and the reflection API. In fact, JavaTypes implementation for reflection is a *Decorator* design pattern implemented over the reflection API.

The two other options of creating *JavaTypes* – bytecode inspection and custom creation – are more interesting.

When creating the *representation from bytecode*, we use the ASM and BCEL bytecode analysis frameworks. ASM [3] provides a Visitor pattern approach for accessing all parts of class data. We have implemented visitors for the particular JavaTypes classes. BCEL [6] is used for historical reasons (introduced earlier than ASM to the project).

The following example illustrates the creation of a class representation from bytecode. Let us have an *Example* class with one method:

```
public class Example {
        public void callMe(int i, String s) { ... }
}
```

The bytecode method descriptor for the method is:

```
(ILjava/lang/String;)V
```

This bytecode data are read by ASM into our *JMethodVisitor* implementation in the *JTypeLoader*. When called, the visitor creates a *JMethod* instance which the type loader adds to a *JClass* object, created earlier in a similar way.

*Custom creation* of JavaTypes can be useful in three cases. The simplest one is testing purposes, when we need to create artificial types to perform tests on the representation. Our second use case is programmatic creation of nonexisting types – this will be described in detail below. Lastly we can use the custom creation principle for *stubbing* purposes. Stubbing is a technique for obtaining the replacement of a class we cannot or do not want to load. Such a class is replaced by a dummy one called *stub*.

The creation of stub *JClass* objects in the custom JType loader is parametrized by a classname mask which defines the set of stubbed classes (e.g. `java.lang.*` for core Java classes). In case of loading via reflection, stubs are created for all classes available on system classpath – the assumption is that those classes are shared by all components in the system and therefore do not influence component substitutability. Stubs are also created in all cases when a class is not available inside the component and we can safely assume that its source is not changed by component update (i.e. that the old and new version of the component will reference the same class code) – this is the case of library classes or imported packages.

### *3.2 Obtaining Complete Bundle Type Representation*

To be able to compare bundles, we have to create the representation of those bundles. This is performed in three steps. The first one is reading the bundle metadata information, in the second and third steps we follow the pointers from this metadata and go to bundle implementation to get the Java layer representation.

#### *3.2.1 Component Metadata – First Step*

Bundle manifest file acts as the point of first contact where the names of packages and other features are found. Bundle layer representation is built from this infor-

mation. This step is trivial, because it means parsing a well specified text file (see example below).

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

The next steps are more interesting, because the JavaType representation must be loaded from bytecode saved in the bundle jar file.

### *3.2.2 Exported Side of Component – Second Step*

The classes for all exported features of a component must be naturally included in the component package itself. We can therefore construct their representation directly from the bytecode of the corresponding types.

The type reconstruction starts at the bundle level. For each exported package and service we create the corresponding *JPackage* or *JService* objects. Then, their *JClass* contents needs to be filled in. The situation is trivial for the service case when only one class (the service interface) is referenced. For packages, the list of all contained classes is first obtained by querying the classloader and then expanded by creating *JClasses* using the reflection type loader.

Next we have to create the representation of all types referenced by public methods or fields of these classes because they will be used in the type-based bundle comparison. This process is bootstrapped by adding the *JClasses* from exported packages and services to the *knownTypesList* queue. Then an iterative algorithm for creating the whole transitive closure of interface types starts. All unprocessed types from *knownTypesList* are handled consecutively. For each type $T$ from *knownTypesList*, the *JTypes* referenced by its members are retrieved. For each type $R$ from these referenced types one of these possibilities is true:

- $R$ is contained in the component and not in *knownTypesList* – add $R$ to *knownTypesList*.
- $R$ is contained in the component and already in *knownTypesList* – no action.
- $R$ is not inside the component and its namespace is listed in an `import-package` header – a stub is created.
- $R$ is not inside the component and its namespace is not in an `import-package` header – exceptional state.

When all referenced types of type $T$ are processed, $T$ is marked as unfolded and next type in the *knownTypesList* is processed with the same algorithm. The exceptional state is handled by throwing the appropriate exception, to indicate that the analysed bundle is invalid (referencing a type in code without corresponding imported package declaration means the bundle would not be resolved and started by the OSGi framework anyway).

Because there is a danger of recursion in type dependencies, the algorithm must include a recursion detection instrument. For this purpose an additional stack data

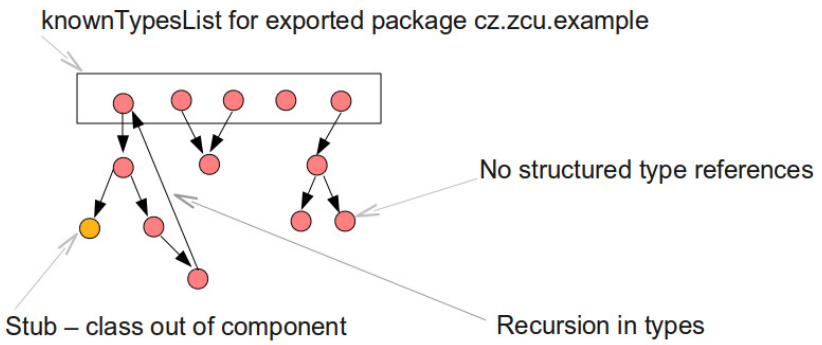



Fig. 6. Exported package – example

structure is used which saves the currently unfolded type branch sequence. On its bottom there is the *JClass* directly referenced from an `export-package` component header. The stack contains the path from the currently processed node to this root. When a new referenced type is found, the stack is checked for containment of this type. If found, recursion was detected and the type is not expanded but the reference is pointed to the stacked instance.

In this way the whole tree of all *JTypes* is expanded. The tree is created because each *JClass* can reference another *JClass* as its field, method parameter, or method return type. The leaf nodes of this tree are of the following three kinds:

- Primitive types. In this trivial case there is no need to create children *JClasses*.
- Stub class. It means this type is contained in another bundle or library, and an empty stub class is created in its place.
- Reference to a recursively defined type. In this case this node in not a leaf, but the expansion of types ends.

### *3.2.3 Imported Side of Component – Third Step*

The situation with the imported side of component is trickier. While for the features on the exported side a concrete bytecode of their types is available within the component package, the situation is the exact opposite for the imported side. This is an obvious consequence of the component-based decomposition of application functionality, as discussed in Section 2.

Our solution is to use the following approach to reconstruct the imported side. Each language type $a$ imported by the component $C$ which is really needed by its functionality is used by at least one type $r$ inside $C$'s implementation. The compiler leaves an imprint of $a$ in the bytecode of $r$ containing the type signatures of the class as well as its members (fields and methods) used by $r$. When we create a union of all these signatures in $C$'s implementation, we get the complete structure of the $a$ type as needed by the whole component.

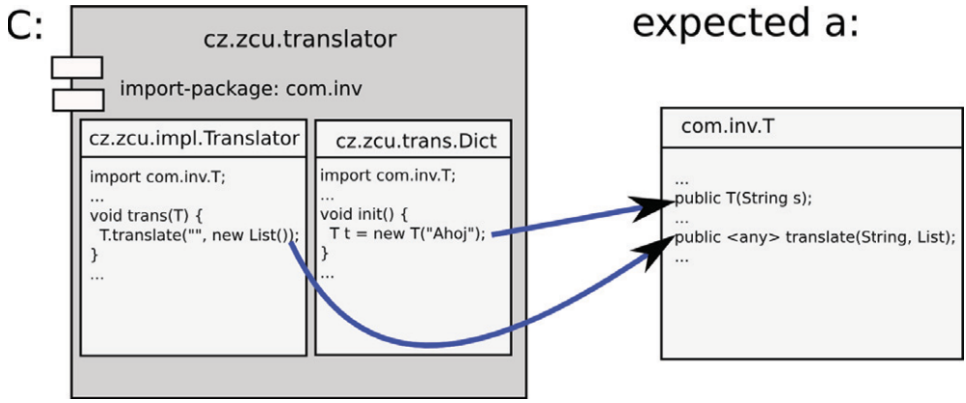



Fig. 7. Imported package – example

This idea can be used to create the representation of imported packages and services, because they are compounds of types. The principle is illustrated by Figure 7. In this example, the component $C$ is *cz.zcu.translator* which imports the package *com.inv*. This package contains a referenced class *com.inv.T*, which is used by two types (*Translator* and *Dict*) in component $C$. The structure of the type $T$ reconstructed from component's implementation consists of the two methods deduced from the code snippets shown in the figure. The symbol ⟨any⟩ denotes any object type, primitive type or void; it acts as a supertype for all types (the calling convention in the bytecode does not contain enough information to reconstruct the precise return type of the operation's signature).

Using this bytecode analysis technique, a similar structure as for exported packages is created. However, the root classes of the representation are created by custom creation (stubbing) described in the previous section.

In certain scenarios it is possible to use an alternative approach to reconstructing imported types. When the bytecode of bundle's imported packages (e.g. their `.jar` files) is available during bundle analysis, we can reconstruct the representation from its "classpath". For instance, when the bundle imports the package `cz.zcu.example` we will include all classes from this package in the representation of bundle's imported side. In section 4 below we show that the assumption of available "classpath" is fulfilled for a large class of situations.

The first approach to obtaining imported types representation is more laborious but exactly matches the component's real requirements. As such it actually provides more precise information than the representation created by analysing the imported packages themselves (the second approach). This advantage is used by the contextual compatibility evaluation [7] proposed earlier by one of the authors.

## 4 Component Compatibility Determination

The method of determining component compatibility we propose is based on evaluating the subtype relation between two components. Briefly, if type $A$ can be used in all possible contexts of another type, then $A$ is a subtype of the other one.

In this section we describe the algorithm of component type-based comparison. Because the focus of this paper is on type representation reconstruction, only the main principle will be illustrated through an example. We first describe a function used to compare type structures, then show some typical use cases for this method.

### *4.1 Component Type Differences*

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:

- $None$: No change between $a$ and $b$.
- $Spec$: Specialization – $b$ is subtype of $a$.
- $Gen$: Generalization – $a$ is subtype of $b$.
- $Mut$: Mutation – there is no subtype relation between $a$ and $b$.

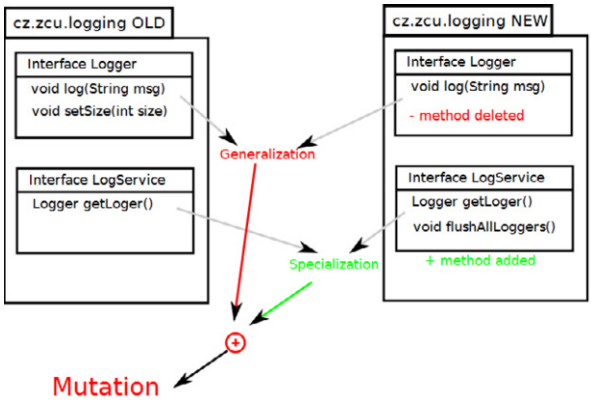



Fig. 8. Subtyping example

The value of $Diff()$ function for structured types is computed by combining the differences of their constituent parts. The exact algorithm of $Diff()$ value determination was published in [5] and is explained in Figure 8 where the package `cz.zcu.logging` is in two versions.

In the second version one method (`void setSize(int)`) in interface `Logger` was deleted and at the same time one method (`void flushAllLoggers()`) of another interface `LogService` was added. Whereas the `Logger` was generalized the `LogService` was specialized. These two changes in the same package are contravariant, so that the resulting difference is a Mutation of the type.

### *4.2 Differences and Compatibility*

When we retrieve the value of $Diff(a, b)$ function we can use it to make a decision about $a$ to $b$ compatibility. The following table (1) shows the rules:

| *Diff* $(a, b)$ | *None* | *Specialization* | *Generalization* | *Mutation* |
|---|---|---|---|---|
| Type $a$ is compatible to $b$ | Yes | No | Yes | No |
| Type $b$ is compatible to $a$ | Yes | Yes | No | No |

Table 1
Mapping of Difference values to Compatibility

### *4.3 Use Cases of The Method*

The method described above is general – it "only" defines how to create representation of Java language types and how to use it in subtyping comparison to determine the level of type and component compatibility. In this section we provide a list of use cases in which the method is used now.

**Automated Versioning**

As described in detail in previous work [5], the bundle comparison method can be used for automated versioning of components. This process can simplify the error-prone task of assigning version identifiers to components and their features. The type differences described above can be used as an input to an automatic creation of version identifiers describing the real evolution of component interface. In the case of OSGi for example, the version numbering scheme is governed by rules which nicely map to the difference values. When using such automated versioning in a component system, its administrators can rely on the reliability of the version identifiers.

This use case applies the method at the bundle release time, when bundle type representations obtained by bytecode inspection of two last component revisions are compared. The first bundle in the comparison is the last previously released component with version identifier. The second bundle is the next release candidate for which we want to determine the version identifier.

With this approach released bundles carry version identifiers which describe not only the piece of software itself but also the changes it has undergone.

**Safe Update**

Another use case of the method is applicable at the deployment time of components. In this case we can use the subtyping comparison to ensure that the new version of a bundle is compatible with the previous one, regardless of the version numbers assigned to both (taking the conservative stand that their reliability is low and that a robust method is needed to ensure application type consistency). Alternativelly, the method can be similarly applied to comparing a new version to the actual context of its deployment.

In this use case the method is applied at component deployment time. The representation of the old bundle version is obtained from reflection, including the imported side (resolved to existing package exporters). The representation of the new bundle version is obtained via bytecode analysis.

When using Safe Updater as an updating tool we can prevent the situation from introductory example in Section 2.4. The Safe Updater searches the interfaces of providers and importers recursively and verifies if types referenced by these two sides of the contract are compatible. The subtyping rules applied recursivelly guarantee those errors preclusion.

In this concrete use case we prevent nearly the same set of errors as would be found during JVM linking and verification processes. Contrary to them we can do these checks on demand without loading the bytecode to JVM.

**Component Dependency Resolving with Checks**

The last application scenario we mention is the process of resolving the components with additional subtyping checks. Resolving, similar to the linking stage of a compilation process, is used to bind imported packages to corresponding exporters. These bindings are made only on matching names of imports and exports.

The additional checks ensure that all mutual component interfaces in a component system are compatible with each other. Here, our method is applied at the start time of the component framework, after component installation or update. Both compared interface sides are loaded via reflection.

# 5 Related Work

In both research and industry world bytecode analysis and manipulation techniques are common. The ASM library [11] can be used to modify existing classes or dynamically generate classes and it is focused on simplicity of use and performance. Other frameworks with similar functionality are JMangler [15] or JavaAssist [10]. Classes are represented by objects which contain all the symbolic information of the given class: methods, fields and byte code instructions, in particular. This approach is less efficient than ASM visitor design pattern, thus ASM is the best choice for dynamic systems.

A distinct class of frameworks uses XML for bytecode representation and manipulation [17,19]. They are comparable in features to the above approaches and support advanced operations including crosscompilation. The latter work discusses the need to transform or wrap API calls embedded in the bytecode.

Unlike JavaTypes, none of these approaches deals with the problems of reconstructing the referenced types not found in the analysed bytecode, which is a key need in the CBSE context. Another advantage of JavaTypes is the ability to compare the reconstructed types by subtyping rules. On the other hand, JavaTypes is not intended for bytecode manipulation and intentionally supports only a limited subset of Java language features.

Concerning the evaluation of component compatibility, there are two general methods. Dynamic assessment determines compatibility by running a regression test suite [12]. More closely related to our approach, McCamant et al [16] define compatibility based on observed (not declared) behaviour while Chaki et al [9] verify that global correctness properties are preserved through component updates, apply-

ing model checking on abstractions of component's source code. These approaches are certainly more precise than compatibility based on type reconstruction used in our work. On the other hand it is much more difficult to obtain the required behavioural representations of a component. Our method could be used as a first check prior to expensive model checking is performed.

## 6 Conclusion

The ability to perform type-based compatibility checks is important for enhanced robustness of component applications. In this paper, we have described a supporting representation of the Java language types which constitute the interface of components together with a set of methods for obtaining this representation. Our system allows to use a mixed set of sources in these methods, including the Reflection API and bytecode analysis using the ASM tool.

Among the main challenges which our approach addresses are (i) the need to cover various stages of component development lifecycle – build, deployment, as well as runtime checks; (ii) limited access to some of the classes referenced by component's interface types; (iii) faithful reconstruction of types of the imported (required) features from a standalone component package. The key contribution presented is the method for obtaining the real structure of the imported-side types from their parts referenced by the component's bytecode implementation.

The methods described in this paper have been successfully used in several applications dealing with component representation and compatibility. They form a base for automated component versioning as well as a type-safe update mechanism implemented for the popular OSGi framework.

## References

[1] Allen, E., *Diagnosing java code: The case for static types* (2002).
URL http://www.ibm.com/developerworks/java/library/j-diag0625.html

[2] *Apache felix web console.*
URL http://felix.apache.org/site/apache-felix-web-console.html

[3] *Asm website.*
URL http://asm.ow2.org/

[4] Bachmann, F. et al., *Volume II: Technical concepts of component-based software engineering*, Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University (2000).

[5] Bauml, J. and P. Brada, *Automated versioning in OSGi: a mechanism for component software consistency guarantee*, in: *Proceedings of Euromicro SEAA* (2009).

[6] *Bcel website.*
URL http://jakarta.apache.org/bcel/

[7] Brada, P., "Specification-Based Component Substitutability and Revision Identification," Ph.D. thesis, Charles University in Prague (2003).

[8] Brada, P., *Enhanced OSGi bundle updates to prevent runtime exceptions*, in: *Proceedings of the 34th Euromicro SEAA conference* (2008).

[9] Chaki, S., E. Clarke, N. Sharygina and N. Sinha, *Verification of evolving software via component substitutability analysis*, Formal Methods in System Design **32** (2008).

[10] Chiba, S. and M. Nishizawa, *An easy-to-use toolkit for efficient java bytecode translators*, in: *Proceedings of the 2nd international conference on Generative programming and component engineering*, Springer-Verlag, New York, NY, USA, 2003, pp. 364–376.

[11] E.Bruneton, R. Lenglet and T. Coupaye, *Asm: a code manipulation tool to implement adaptablesystems*, in: *Adaptable and extensible component systems*, Grenoble, France, 2002.

[12] Flores, A. and M. Polo, *Testing-based process for evaluating component replaceability*, in: *Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008)*, 2009, pp. 101 – 115, Electronic Notes in Theoretical Computer Science, vol. 236.

[13] Forman, I. R., N. Forman, D. J. V. Ibm, I. R. Forman and N. Forman, *Java reflection in action* (2004).

[14] James Gosling, G. S., Bill Joy and G. Bracha, *Java language specification, third edition* (2005).

[15] Kniesel, G., P. Costanza and M. Austermann, *Jmangler-a framework for load-time transformation of java class files*, in: *IEEE International Workshop on Source Code Analysis and Manipulation* (2001).

[16] McCamant, S. and M. D. Ernst, *Formalizing lightweight verification of software component composition*, in: *Proceedings of SAVCBS 2004: Specification and Verification of Component-Based Systems*, Newport Beach, CA, USA), 2004, pp. 47–54.

[17] NoUnit Team, *NoUnit* (2006), accessed 12/2009.
URL http://nounit.sourceforge.net/

[18] The OSGi Alliance, "OSGi Service Platform, Release 4," (2005), available at http://www.osgi.org/.

[19] Puder, A. and J. Lee, *Towards an XML-based bytecode level transformation framework*, in: E. Albert and S. Genaim, editors, *Preproceedings of 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, York, UK, 2009.

[20] Szyperski, C., "Component Software, Second Edition," ACM Press, Addison-Wesley, 2002.