Software Components Compatibility Verification Based on Static Byte-Code Analysis

Kamil Jezek  
NTIS – New Technologies for the Information Society  
European Centre of Excellence  
Faculty of Applied Sciences  
University of West Bohemia  
kjezek@ntis.zcu.cz

Lukas Holy, Antonin Slezacek, Premek Brada  
Department of Computer Science and Engineering  
Faculty of Applied Sciences  
University of West Bohemia  
{lholy,brada}@kiv.zcu.cz

Euromicro SEAA 2013
Agenda

- Compatibility problems of current component based applications
- Proposed solution: Dependencies reconstruction and evaluation
- Classification of incompatibilities that can be discovered
- Approach implementation
- Case-study
- Limitations
Problem

Applications from preexisting components may contain type incompatibilities
• Type compatibility traditionally ensured by compilation
• Preexisting components already compiled
• Compile time checks skipped
• Other means to ensure type compatibility required

Type errors manifested at runtime
• e.g. LinkageError in Java
• NoSuchMethodError
• ClassNotFoundException
Proposed Solution

• Dependencies reconstruction from binaries
  – Byte-code in Java

• Dependencies' consistency evaluation
  – In terms of binary compatibility

• Compatibility decision
  – Prevents runtime errors
  – “as if compiled by compiler” but working with binary compatibility (differ in Java)

  Created Java byte-code analyzer an verification tool
Incompatibilities Classification

- **Missing dependencies (1)**
  - Produce runtime exceptions (NoSuchMethodEx, ClassNotFoundEx, ...)
  - e.g. missing libraries, forgotten transitive dependent library

- **Inconsistent dependencies (2)**
  - Produce runtime exceptions (LinkageError, ...)
  - e.g. incompatible method signatures caused by wrong library version

- **Redundant dependencies (3)**
  - No error, but increased application size
  - e.g. forgotten test libraries in production code (JUnit)

- **Duplicated dependencies (4)**
  - Application may misbehave (two libraries provide the same API, differ in function)
  - e.g. two different implementations of logging
Dependency Reconstruction

Java byte-code analyser

Component API (e.g. CGlib)

```java
public class Enhancer {

    public static void registerCallbacks(
        Class generatedClass,
        Callback[] callbacks) { … }

    public static void registerStaticCallbacks(
        Class generatedClass,
        Callback[] callbacks) { … }

    private Class<?> createClass(
        Enhancer enhancer) {
        Class<?> subclass = enhancer.createClass();
        Enhancer.registerStaticCallbacks(
            subclass, this.callbackInstances);
        return subclass;
    }
}
```

API invocation (e.g. Spring)

```java
class ConfigurationClassEnhancer {

    private Class<?> createClass(
        Enhancer enhancer) {
        Class<?> subclass = enhancer.createClass();
        Enhancer.registerStaticCallbacks(
            subclass, this.callbackInstances);
        return subclass;
    }
}
```
Dependency Evaluation

- Two models compared
  - Missing dependencies (1)
  - Inconsistent dependencies (2)
  - Redundant dependencies (3)
  - Duplicated dependencies (4)

API invocation

Component API
Compatibility Decision

Redundant dependencies (3)

JClass

Inconsistent dependency (2)

Duplicated dependencies (4)

Missing dependency (1)
Case-study

- Analysed JEE application
  - Spring, OrientDB, FreeMarker, ... (50 libraries)
  - Build by maven
- Developed Maven plugin to run evaluation
  - Invoked on build (verify phase)
- Discovered problems:
  - Inconsistent dependency: wrong CGlib version (v2.1 instead of v2.2)
    - Caused runtime exception
  - Duplicated dependency: logging: jcl-over-slf4j and commons-logging
    - Required two configuration files
  - Redundant dependency: JUnit in production code
Limitations

- Approach limited to static types in permanent byte-code
  - Analyses also API invocations in code that is never invoked (death code)
  - Byte-code generated at runtime, Java Reflection ignored
    - But common in current Java frameworks
- In Case-study
  - Migration to Tomcat 5.5
    - runtime JSTL definition loading not detected (NoClassDefFoundError:javax/el/ValueExpression)
  - Missing JSON parser
    - runtime classpath scanning not detected (exception on AJAX request invocation)
Conclusion

- Static incompatibilities not detected by compilers in current software
- Proposed byte-code analyser to fill this gap
- Implemented as Maven plugin
- Shown on JEE application
- Usable in development process
  - By Maven plugin
Thank you for your attention

Questions?

Find us:
Static compatibility tool (JaCC)
https://www.assembla.com/spaces/jacc/
Maven plugin (CCP3)
https://www.assembla.com/spaces/ccp3/

Contact us:
kjezek@ntis.zcu.cz, brada@kiv.zcu.cz, lholy@kiv.zcu.cz