2012 19th Asia-Pacific Software Engineering Conference

Semi-static Detection of Runtime Type Errors in Component-based Java Programs

Xiaowei Zhou[1,2], Wenbo Zhang[1], Jianhua Zhang[1,2]
1. Technology Center of Software Engineering, Institute of Software,
Chinese Academy of Sciences, Beijing 100190, China
2. Graduate University, Chinese Academy of Sciences, Beijing 100190, China
{zhouxiaowei08, zhangwenbo, zhangjianhua07}@otcaix.iscas.ac.cn

***Abstract*—The using of multiple custom class loaders in component-based Java programs may lead to more runtime type errors. These errors can happen at various program statements and may be wrapped in different types of exceptions by JVM, therefore posing difficulties for dealing with them. Traditional static analysis approaches only consider static types and thus cannot detect many of them. We propose a semi-static approach based on points-to analysis and dynamically gathered behavior information of Java class loaders to detect runtime type errors in component-based Java programs without running them. We also implement a prototype tool for OSGi-based programs, where OSGi is a typical Java component framework.**

Keywords-Component-based; Points-to Analysis; Class Loader; Semi-static Detection

I. INTRODUCTION

In Java, type checking is done both statically and dynamically. Type-related defects escaped from Java compilers and detected by JVM's runtime type checking are called *runtime type errors* [1] (e.g. those caused by unsafe casts).

With component-based software development (CBSD) being increasingly adopted as a mainstream approach of software engineering [2], a lot of Java programs also adopt the component-based paradigm, such as web applications based on Servlet/JSP, enterprise applications using EJBs, OSGi[1]-based programs and so on.

Runtime type errors are more frequent in component-based Java programs. First, in component-based Java programs, *component containers*, like web application servers and OSGi frameworks, usually create several custom class loaders and different application classes may be *defined* [3] by different class loaders at runtime. For example, in OSGi-based programs, classes in each *bundle* (OSGi-compliant component) are defined by the bundle's class loader [4]; and in Java web systems, classes in a web application are defined by its dedicated class loader and those classes of the application server are defined by other class loaders. In this paper we only consider this kind of component-based Java programs where classes in different components are defined by different class loaders. Second, it is likely that in a component-based Java program, components may contain *same-named classes*[2]. For example, JOnAS 5.2.0, an OSGi-based Java EE application server, has 77 bundles which are active in execution, and there are 105 distinct class names[3] of which each is owned by more than one class. Same-named classes usually result from the wide use of application frameworks and third-party libraries. The instance of a class or its subclass created in one component may propagate to the code of another component which contains a class of the same name. Since same-named classes in different components are defined by different class loaders and thus these classes represent different *runtime types* [1], this propagation may cause some reference variables to point to objects with wrong runtime types. This kind of defects cannot be detected by compilers since they only consider static types, whereas JVM will detect these runtime type errors and raise exceptions like *ClassCastException*, *VerifyError* and *ArrayStoreException*.

Allowing only one of the same-named classes to be loaded will prevent this kind of errors, possibly by deleting classes until there is only one class left for every class name. However, this approach has a problem. Same-named classes may have different code, for they may come from different versions of third-party libraries, and each of them may have a set of static fields which will all take effect if all of these classes are loaded. If only one of these classes is permitted to be loaded, the semantics of the program may be damaged. Avoiding such errors by coding standards or best practices, for example, prohibiting the instances of same-named classes (or their subclasses) to be propagated beyond their own components, is also impractical. First, which classes are same-named classes usually are not known before the components are integrated. Second, components may come from various vendors and letting these vendors comply with the same coding standard is difficult. Third, some third-party components are somewhat casually migrated from legacy code, such as the official OSGi-compliant log4j 1.2.16, which *imports* and *exports* [4] so many Java packages that the clarity of component interface is damaged. Exceptions caused by runtime type errors may occur at 'almost any position' of the program, rendering writing exception handling code to recover from these errors very hard.

Statically detecting runtime type errors will help programmers find out faults at early stage and make corresponding remedies. There have been some works using static analysis to detect runtime type errors caused by unsafe casts [5-8]. However, these works do not consider runtime

[1] http://www.osgi.org
[2] May also be called *duplicate classes*.
[3] When we mention *class name*, we mean the fully qualified name, which includes the name of the Java package containing the class.

1530-1362/12 $26.00 © 2012 IEEE
DOI 10.1109/APSEC.2012.46

IEEE computer society

type discrepancies caused by class loaders and thus cannot detect many of the runtime type errors.

We propose a semi-static approach based on points-to analysis [9] to detect runtime type errors in component-based Java programs. We invoke class loaders provided by component containers to get their behavior and then we are able to figure out the defining class loader of the *allocation sites* [9]. Then the runtime types of objects a reference variable may point to can be obtained. Also, we can easily acquire the runtime types of the reference variables given class loaders' behavior information. Given these runtime types, we check every program statement where JVM may raise exception [3] due to runtime type error, for the possibility that related variables point to wrong-typed allocation sites.

We use OSGi as the example of Java component model and framework. OSGi is so widely used that it is emerging as the de facto dynamic module system for Java applications [10], and runtime type errors are relatively common in developing OSGi-based programs. We implement our approach into an open-source prototype tool and show some results of the case studies.

The remainder of this paper is organized as follows. Section II analyzes the problem of runtime type errors in component-based Java programs. Section III presents our approach. Section IV talks about some issues on the implementation of our tool. Section V presents case studies. Section VI discusses related work. Section VII concludes the paper and gives future work.

II. ABOUT THE PROBLEM

A. Background

1) Class Loading and Linking in Java

In Java, all the classes are loaded into JVM by *class loaders* [3] at runtime. Class loaders are also Java objects (except for the *bootstrap class loader* provided by JVM which is used to load some core classes of Java Runtime Environment). A class loader may *delegate* to another class loader to look for a class; after several (may be zero) *delegations*, one class loader will finally load the class by itself. The class loader which is requested for loading a class (by passing the class name as parameter) is called the class's *initiating class loader*, and the one which loads the class by itself after delegations is called the *defining class loader* of this class; the two class loaders may be the same or not. A *runtime class* is identified by its class name and its defining class loader, therefore two runtime classes must not be the same if they have different defining class loaders, even if they had the same name or were created from the same class file.

Java programmers may create their own custom class loaders, and component containers usually also create several class loaders for themselves and for components hosted by them, so in a Java runtime environment, there may exist several class loaders besides those provided by JVM.

A class usually has a lot of *symbolic references*[4] [3] to other classes, such as its super class, classes included in its field types and the classes referred to in the code of its methods and so on. The defining class loader of one class will initiate the loading of these referred to classes when needed.

2) OSGi

We take OSGi as our case of Java component model and framework. An OSGi-based program consists of several bundles interacting with each other. Fig. 1 shows the architecture of an OSGi-based program. One bundle can only use some of the classes of other bundles. For example, if a bundle wants to create an instance of a class in another bundle, the using and providing of the class should be explicitly declared in the meta data of corresponding bundles. This is guaranteed by OSGi's class loading mechanism. OSGi framework provides every bundle a dedicated class loader, which will define all the classes in the corresponding bundle. The OSGi specification specifies the workflow of these class loaders.

B. An Example

Here is a simple example to show runtime type errors in an OSGi-based program. We create two bundle projects in Eclipse, named "Test1" and "Test2", as shown in Fig. 2. Both bundles contain the class named "base.Base" where "base" is its package name, and "Test2" contains another "test2.Derived" class which is a subclass of "base.Base".

The bundle "Test1" will invoke the class of "test2.Derived", so we add "Import-Package: test2" to the meta data (MANIFEST.MF file) of "Test1", and "Export-Package: test2;uses:="base"" to that of "Test2". Some codes of the "Test1" is shown in Fig. 3.

Interestingly, the two bundles can be successfully compiled but will get a *java.lang.VerifyError* when running. This is because the variable *baseVar* points to a wrong-typed object when being used in field reference "baseVar.field" in

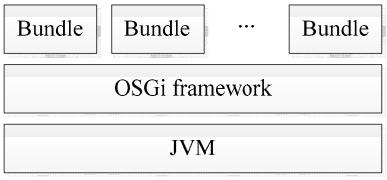

Figure 1. OSGi-based program

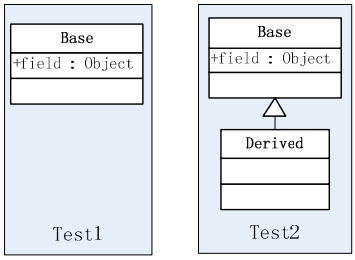

Figure 2. Example bundles Test1 and Test2

[4] These references only provide names of the classes they refer to, so they are 'symbolic'.

```
15:  public void testFieldRef() {
16:      Base baseVar = new Derived();
17:      baseVar.field = new Object();
18:  }
```

Figure 3. Some code in Test1 bundle

Fig. 3. At runtime, "test2.Derived" is the subclass of "base.Base" defined by the class loader of "Test2" bundle, but the type of the variable *baseVar* is "base.Base" defined by the class loader of "Test1" bundle.

In real-world programs, object reference has many ways of propagation, and it may be passed on many times until a runtime type error will occur.

C. Problem Analysis

When a component-based Java program is running, classes in different components may interact with each other. For example, a class instance of one component may hold an instance of a class of another component. When interacting, some variables may be assigned wrong-typed object references, and runtime type errors may occur.

In fact, even for some traditional Java programs where all the application classes are defined by the system class loader provided by JVM, there are also chances of runtime type errors, such as that caused by unsafe casts. However, for some component-based Java programs, the causes of runtime time type errors are more complex. At runtime, classes in different components are defined by different class loaders, and some components may contain same-named classes. Then it may happen that some variables are attempted to be assigned wrong-typed object references which have correct static types, e.g. the example in section II.B. Runtime type is decided by its corresponding static type and the defining class loader of the class involved in that static type. Static types do not distinguish between same-named classes, i.e. considering them to be identical, and it is also the case when figuring out subclass relationships. Java compilers only check static types, so the code in section II.B will pass the compilation. However, when running, since the factor of the defining class loader is introduced, same-named classes in different components will not be identical and some statically valid subclass relationships will no longer be valid. This can be viewed as the criteria for well-typedness are more strict when considering runtime types. And this kind of runtime type errors is caused by the "dynamic part" (defining class loaders) of runtime types.

The class loading scheme of Java, especially that of OSGi, is somewhat difficult for programmers to master and correctly use, just like exception handling and multi-threading. When writing component-based programs, some Java programmers may casually believe that same-named classes represent the same type, causing runtime type errors easier to happen.

Whether there are runtime type errors partly depends on the behavior of the class loaders. For example, which runtime class will be the superclass of a runtime class is decided by the latter's defining class loader, and whether two runtime classes have subclass relationship may determine whether there will be runtime type errors. The behavior of custom class loaders is undecidable. Therefore, Java compiler or static analysis cannot detect runtime type errors caused by defining class loaders generally. However, the behavior of class loaders provided by component containers are not arbitrary. For example, the OSGi specification specifies a class loading mechanism, and servlet container Tomcat also has a description of its class loaders[5]. Generally, in fact, component containers are so well tested (by users) that we can believe their class loaders are terminable and deterministic, so their behavior can be known without executing the whole program, making statically checking for runtime type errors in component-based Java programs feasible.

III. Static Detection of Runtime Type Errors

A. Points-to Analysis

Static detection of type errors in a program can be conducted by checking whether reference variables in the program may point to objects which do not have correct types, using points-to analysis. Some work uses points-to analysis to check the safety of casts [5].

Points-to analysis for Java computes a points-to relation that maps each reference variable to a superset of the objects that it may point to during execution. In points-to analysis, object is usually abstracted to *allocation site* (the location of "new" statement for creating this object). When a program is running, an allocation site may be passed several times by the execution trace and several objects may be created, but these objects are of the same type. Thus, the abstraction of objects to allocation sites will satisfy the need of checking for type errors.

Previous points-to analysis approaches only consider static types, so they cannot detect runtime type errors caused by the "dynamic part" of runtime types. To detect this kind of errors, we also need to consider defining class loaders in points-to analysis.

We first describe Andersen's style [11] points-to analysis for Java [12], and then present how to add runtime type to the analysis. We define three sets: R contains all the reference variables in the program under analysis, O contains all the allocation sites, and F contains all the fields of the program's classes. The analysis process will gradually construct a points-to graph; a points-to graph example is shown in Fig. 4 [12]. Points-to graph contains two kinds of edges: edge $(r, o_i) \in R \times O$ denotes variable r may point to allocation site o_i; edge $(\langle o_i, f \rangle, o_j) \in (O \times F) \times O$ denotes the field f of allocation site o_i may point to allocation site o_j.

Some statements in the program under analysis may cause the propagation of object references which is depicted by the transition function f. Table I shows the transition functions for 5 common statements. In these functions, G is the points-to graph (its edge set); Pt is the function which retrieves the set of pointed to allocation sites (*points-to set*) for reference variable or field of allocation site; the function $dispatch$ is to determine which method will be actually called given an allocation site pointed to by the target

[5] http://tomcat.apache.org/tomcat-6.0-doc/class-loader-howto.html

TABLE I TRANSITION FUNCTIONS OF ANDERSON'S STYLE POINTS-TO ANALYSIS

Statement	Transition function
Object creation: $l = new\ CN$	$f(G, s_i : l = new\ CN) = G \cup \{(l, o_i)\}$
Direct assignment: $l = r$	$f(G, l = r) = G \cup \{(l, o_i) \mid o_i \in Pt(G, r)\}$
Field writing: $l.f = r$	$f(G, l.f = r) = G \cup \{(\langle o_i, f\rangle, o_j) \mid o_i \in Pt(G, l) \wedge o_j \in Pt(G, r)\}$
Field reading: $l = r.f$	$f(G, l = r.f) = G \cup \{(l, o_i) \mid o_j \in Pt(G, r) \wedge o_i \in Pt(G, \langle o_j, f\rangle)\}$
Virtual call: $l = r_0.m(r_1,...,r_n)$	$f(G, l = r_0.m(r_1,...,r_n)) = G \cup \{resolve(G, m, o_i, r_1,...,r_n, l) \mid o_i \in Pt(G, r_0)\}$ $resolve(G, m, o_i, r_1,...,r_n, l) =$ **let** $m_j(p_0, p_1,..., p_n, ret_j) = dispatch(o_i, m)$ **in** $\{(p_0, o_i)\} \cup \{f(G, p_1 = r_1)\} \cup ... \cup \{f(G, l = ret_j)\}$

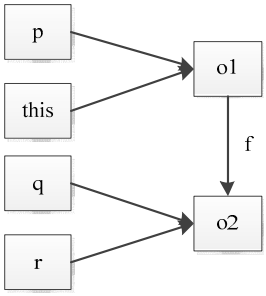

Figure 4. Points-to graph
(*p, this, q, r* are reference variables, *o1, o2* are allocation sites, *f* is a field of *o1*)

variable of a virtual call statement and a method signature. Most of these transition functions have a transparent meaning. For example, for the direct assignment case, the transition function creates *points-to edges* from l to all the allocation sites pointed to by r; and for the virtual call case, the transition function depicts the propagation of allocation sites via parameters and return values. Other statements, like static method calls, can be treated analogously. Furthermore, some complex statement may be equivalently transformed into several simpler statements to fit transition functions, for example, $l.f_1 = r.f_2$ can be transformed to $v = r.f_2$ followed by $l.f_1 = v$, in which v is a temporary variable.

The process of points-to analysis begins with an empty G, and then iterate over program statements, replacing G with the result of the transition function, i.e. $G \leftarrow f(G, stmt)$, until G stops becoming larger.

B. Getting Runtime Types of Allocation Sites

We invoke class loaders provided by component containers, to get the runtime types of allocation sites.

1) Assumptions

We make two assumptions for the class loaders we pay attention to. First, the loading process initiated by these class loaders must be terminable. Second, the behavior of these class loaders is deterministic, i.e. different invocations of "loadClass" at different time with the same parameter will return the same result (return the same runtime class or raise the same exception).

2) Loading Function

Under these assumptions, we invoke the class loaders to get the behavior of them for static analysis by requesting them to load classes and using AOP techniques to get the resulting defining class loader without actually loading the class into JVM. We then model their behavior as loading function:

$$load : String \times ClassLoader \rightarrow ClassLoader$$

This function gets a class name and a class loader, and returns the defining class loader of the loaded class when the argument class loader initiates the loading process for the class name. If the loading process fails, $null$ will be returned. Bootstrap class loader is not an ordinary Java object, and will be represented as null in JVM; in loading function, we use an object different from any other class loaders to represent bootstrap class loader, to differentiate from the case of loading failure.

3) Extending Points-to Analysis

Same-named classes defined by different class loaders are different, and the reference variables declared in different runtime classes (fields and local variables) are also different. So, we attach a runtime class C to every reference variable to differentiate between variables in same-named classes defined by different class loaders. We attach a defining class loader to every allocation site; with this we will be able to obtain the runtime type of it, since the static type of an allocation site can be easily retrieved. Then the edge (r, o_i) in points-to graph will be extended to $(\langle r, C\rangle, \langle o_i, CL\rangle)$ which denotes the reference variable r in runtime class C may point to allocation site o_i with defining class loader CL; edge $(\langle o_i, f\rangle, o_j)$ will be extended to $(\langle o_i, CL_1, f\rangle, \langle o_j, CL_2\rangle)$ which denotes the field f of allocation site o_i with defining class loader CL_1 may point to allocation site o_j with defining class loader CL_2. An example of extended points-to graph is shown in Fig. 5.

The extension of transition functions is shown in Table II. We add an argument C to transition functions to denote the runtime class in which the argument statement resides. For the object creation case, according to JVM's resolution mechanism [3], the defining class loader of C will be used to find referred classes, so we use loading function to determine the defining class loader of allocation site o_i ($C.defCL$ is the defining class loader of C). The virtual call case involves parameter passing between classes; we make a special transition function f' to deal with this; function

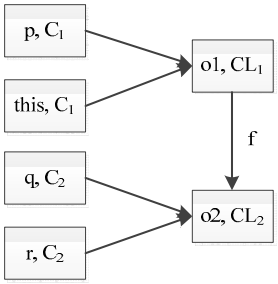

Figure 5. Extended points-to graph

$dispatch$ uses runtime subclass relationship to find the actually called method and gets the runtime class in which the method is declared, which can be easily achieved since loading function can be used to determine a class's runtime super class. We have modified the representation of points-to graph and transition functions; this will not affect the process of analysis. With our extended points-to analysis, the runtime type of allocation sites pointed to by a reference variable can be retrieved.

There is another issue: how to obtain $C.defCL$ in the transition function for the object creation case. For component-based Java programs, this is usually decided by the location of the class file. As to OSGi, according to its specification, the classes in a bundle will be defined by this bundle's class loader.

C. Detecting Runtime Type Errors

With extended points-to analysis, we can check if a variable may point to wrong-typed allocation sites. However, in the actual execution of Java programs, JVM will not raise exception even if some variables points to objects with wrong runtime types. For example, in the code shown in section II.B, if we delete line 17 in Fig. 3, then the program will not raise exception; actually, the previous line of code will cause the *baseVar* variable to point to wrong-typed object. This is the consequence of JVM's bytecode verification mechanism. JVM pays attention to whether a local variable points to a wrong-typed object only at some points [13], for example, when the variable is to be assigned to a field or returned by a method.

We statically simulate the bytecode verification mechanism of JVM by choosing some statements (or fractions of statements) which may cause JVM to raise exceptions to check. This enables us to know which kinds of exception may be raised from the detected runtime type errors and which statements may actually cause exceptions. We first use Java static analysis framework Soot [14] to convert the program under analysis to Jimple [15] intermediate code. Jimple has only 15 kinds of statements, which makes the analyzer program simpler to implement. With Jimple code, there are 6 cases to check for runtime type errors.

1) Instance Field Referring ($v.field$)

Here we check if local variable v may point to wrong-typed allocation sites; we describe the criteria as a logic formula:

$$\exists\langle o_i,CL\rangle(\langle o_i,CL\rangle \in pto(\langle v,C\rangle) \wedge \neg(\langle o_i,CL\rangle : \langle v,C\rangle.type))$$

In which $\langle v,C\rangle.type$ is the runtime type of local variable v in runtime class C (v is declared in C's method); the defining class loader part in the type can be obtained by loading function, i.e. $load(v.classintypename,C.defCL)$, where $v.classintypename$ is the class name included in v's static type. $\langle o_i,CL\rangle : \langle v,C\rangle.type$ denotes $\langle o_i,CL\rangle$ is an instance of $\langle v,C\rangle.type$, resembling the *instanceof* operator in Java. $pto(\langle v,C\rangle)$ is the points-to set of $\langle v,C\rangle$ obtained by the points-to analysis. If the formula evaluates to true, we get a runtime type error and give out a warning.

2) Field Writing ($v_1.field = v_2$)

This case includes writing to instance field and static field, we only take the former as an example here and the latter is alike. Here we check if the well-typedness of $field$ may be spoiled by the assignment. The formula is:

$$\exists\langle o_i,CL\rangle(\langle o_i,CL\rangle \in pto(\langle v_2,C\rangle) \wedge \neg(\langle o_i,CL\rangle : \langle v_1.field,C\rangle.type))$$

In which $\langle v_1.field,C\rangle.type$ is the runtime type of $field$ which is declared in the class included in $\langle v_1,C\rangle.type$.

3) Array Storing ($a[ind] = v$)

Here we check if the well-typedness of array elements may be spoiled by assignment. The formula is:

$$\exists\langle o_i,CL\rangle(\langle o_i,CL\rangle \in pto(\langle v,C\rangle) \wedge \neg(\langle o_i,CL\rangle : \langle a[ind],C\rangle.type))$$

In which $\langle a[ind],C\rangle.type$ is the element type of array a.

4) Type Casting ($(T)v$)

Here we check if the cast may be illegal. The formula is:

TABLE II EXTENDED TRANSITION FUNCTIONS

Statement	Transition function
Object creation: $l = new\ CN$	$f(G,s_i : l = new\ CN,C) = G \cup \{(\langle l,C\rangle,\langle o_i,load(CN,C.defCL)\rangle)\}$
Direct assignment: $l = r$	$f(G,l = r,C) = G \cup \{(\langle l,C\rangle,\langle o_i,CL\rangle \mid \langle o_i,CL\rangle \in Pt(G,\langle r,C\rangle))\}$
Field writing: $l.f = r$	$f(G,l.f = r,C) = G \cup \{(\langle o_i,CL_1,f\rangle,\langle o_j,CL_2\rangle) \mid \langle o_i,CL_1\rangle \in Pt(G,\langle l,C\rangle) \wedge \langle o_j,CL_2\rangle \in Pt(G,\langle r,C\rangle)\}$
Field reading: $l = r.f$	$f(G,l = r.f,C) = G \cup \{(\langle l,C\rangle,\langle o_i,CL\rangle) \mid \langle o_j,CL_1\rangle \in Pt(G,\langle r,C\rangle) \wedge \langle o_i,CL\rangle \in Pt(G,\langle o_j,CL_1,f\rangle)\}$
Virtual call: $l = r_0.m(r_1,...,r_n)$	$f(G,l = r_0.m(r_1,...,r_n),C) = G \cup \{resolve(G,m,\langle o_i,CL_1\rangle,r_1,...,r_n,l,C) \mid \langle o_i,CL_1\rangle \in Pt(G,\langle r_0,C\rangle)\}$ $resolve(G,m,\langle o_i,CL_1\rangle,r_1,...,r_n,l,C) =$ **let** $C_1.m_j(p_0,p_1,...,p_n,ret_j) = dispatch(\langle o_i,CL_1\rangle,m)$ **in** $\{(\langle p_0,C_1\rangle,\langle o_i,CL_1\rangle)\} \cup \{f'(G,p_1 = r_1,C_1,C)\} \cup ... \cup \{f'(G,l = ret_j,C,C_1)\}$ $f'(G,l = r,C_l,C_r) = G \cup \{(\langle l,C_l\rangle,\langle o_i,CL\rangle \mid \langle o_i,CL\rangle \in Pt(G,\langle r,C_r\rangle))\}$

$$\exists\langle o_i,CL\rangle(\langle o_i,CL\rangle \in pto(\langle v,C\rangle) \wedge \neg(\langle o_i,CL\rangle : \langle T,CL_T\rangle))$$

In which $\langle T,CL_T\rangle$ is the runtime type of T, and $CL_T = load(T,C.defCL)$.

5) Method Calling ($v_0.m(v_1,v_2,...)$)

Here we check if the calling target (static method call does not have a target) and parameters may point to wrong-typed allocation sites. The formula is:

$$\exists v\exists\langle o_i,CL\rangle(v \in \{v_0,v_1,v_2,...\} \wedge \langle o_i,CL\rangle \in pto(\langle v,C\rangle) \wedge\neg(\langle o_i,CL\rangle : \langle v,C\rangle.type))$$

6) Method Returning ($return\ v$)

Here we check if local variable v may point to wrong-typed allocation sites. The formula is:

$$\exists\langle o_i,CL\rangle(\langle o_i,CL\rangle \in pto(\langle v,C\rangle) \wedge \neg(\langle o_i,CL\rangle : \langle v,C\rangle.type))$$

IV. IMPLEMENTATION

We implement a prototype tool called Clap[6], which is specific to OSGi-based programs now. This tool is based on Java static analysis framework Soot and OSGi framework Felix[7]. We add a defining class loader field to Java class's representation class "soot.SootClass" and Java reference type's representation class "soot.RefType", and change Soot's code of resolving symbolic references to using loading function to determine the defining class loaders of the referred classes. Loading function can be simulated by invoking the code of a slightly modified version of bundle class loader provided by Felix. Put the bundles under analysis into Felix's deployment folder, adjust Felix's configuration file (set *bootdelegation*s and so on) if necessary, and begin the analysis by just starting Clap. Clap will start the analysis after Felix installs and resolves every bundle, and none of the bundles will start and run during the process. The system composed of Clap and the program under analysis is shown in Fig. 6.

A. Context Sensitivity

There is work showing that context-sensitive points-to analysis is more precise than context-insensitive one [5]. However, the former is more costly. In our analysis, we only consider the situation when the class name included in the right part of ':' operator in section III.C belongs to same-named classes. Since the number of same-named classes are relatively small compared to the total number of classes in a program, we use demand-driven context-sensitive points-to analysis [6], which is efficient when we only need to get points-to sets for a small number of variables.

B. Entry Points

In Soot, points-to analysis should have entry points. For ordinary Java programs, the entry points should contain the 'main' method.

For OSGi-based programs, we choose the methods for activating and deactivating bundles as entry points.

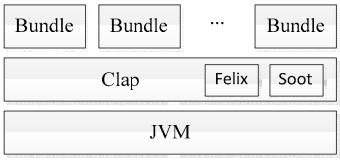

Figure 6. Clap system structure

C. Checking for Loading Constraint Violations

In OSGi-based programs, same-named classes may also cause JVM to raise *LinkageError* because of *loading constraint* violations. Checking for this kind of errors is relatively simple, for it does not even need points-to analysis. In Clap, we implement this checking which also uses defining class loaders of classes and loading function. The principle of this checking is just a static simulation of JVM's process of enforcing loading constraints[3].

D. Checking Algorithm

The pseudo-code description of our checking algorithm is shown in Fig. 7.

V. CASE STUDIES

A. Analyzing the Example Program in Section II.B

We used Clap to analyze the example program in section II.B.; the analysis finished after a couple of seconds on an ordinary PC (Core i7 3.4GHz, 4GB Mem, Windows 7). Clap gave out a runtime type error warning of case "instance field referring", saying the variable *baseVar* at line 17 in Fig. 3 may point to wrong-typed allocation site in the previous line. This warning is correct.

The propagation trace contains 4 edges. The first two edges are allocation edge and assign edge both derived from line 16. These two edges will sufficiently make *baseVar* variable be in error. The last two edges do not have a corresponding source code line number. To better explain this, we should look at the Jimple code of "testFieldRef" method, generated by our modified Soot, in Fig. 8. Our modified Soot will consider defining class loader in inferring types of local variables [13]. Being aware that "test2.Derived" is not a subclass of "base.Base", it inserts a cast statement "r6 = (base.Base) r2;", where "r6" corresponds to variable *baseVar* at line 17 and "r2" to variable *baseVar* at line 16 in Fig. 3, and the last two edges

```
initialize modified bundle class loaders, as loading function
load all the bundles' classes into Soot
figure out same-named classes
perform our extended points-to analysis

for each sootclass c in soot's scene
  for each method m in c
    check loading constraints for method overriding on m
    for each statement stmt in m
      check loading constraints for field reference in stmt
      check loading constraints for method reference in stmt
      check stmt for runtime type errors according to section
III.C
    endfor
  endfor
endfor
```

Figure 7. Checking algorithm

[6] The source code of Clap is available at http://code.google.com/p/clap/
[7] http://felix.apache.org/site/index.html

```
public void testFieldRef()
{
   test1.FormalTest r0;
   test2.Derived r2, r4;
   java.lang.Object r5;
   base.Base r6;

   r0 := @this: test1.FormalTest;
   r4 = new test2.Derived;
   specialinvoke r4.< test2.Derived: void <init>()>();
   r2 = r4;
   r5 = new java.lang.Object;
   specialinvoke r5.<java.lang.Object: void <init>()>();
   r6 = (base.Base) r2;
   r6.<base.Base: java.lang.Object field> = r5;
   return;
}
```

Figure 8. The Jimple code of testFieldRef

are derived by this cast statement.

B. Analyzing a Log4j-related Example

We analyzed an example of log4j's classes existing in two bundles. We used log4j 1.2.16, whose jar file is already a bundle, which we call "Log4jBundle". We made another bundle called "Log4jTest1" and embedded log4j's classes in it. Then log4j's classes exist in both bundles and become same-named classes. We let "Log4jTest1" bundle import the "org.apache.log4j" package but no other log4j packages like "org.apache.log4j.net" and so on. Thus, "Log4jTest1" bundle will use log4j's classes in "org.apache.log4j" package from bundle "Log4jBundle" and those in other packages from itself. "Log4jTest1" bundle has some code shown in Fig. 9, in which class "Logger" and "SMTPAppender" come from different bundles. In addition, we should add another two bundles "javax.mail" and "javax.activation" to make the whole program able to run, and thus the whole program consists of 4 bundles.

Clap ran the analysis on a PC server (Xeon E5620 2.4GHz, 8GB Mem, Ubuntu 11.10 x64) for about 7 minutes and detected 150 runtime type errors and 35 loading constraint violations. In fact, this program will end in a LinkageError due to loading constraint violation, which muffle other errors; this violation has been detected by Clap. From the analysis result, we found that some reference variables in "Log4jTest1" bundle may point to objects created in "Log4jBundle" bundle; we used a simple approach to correct the problem by removing "Log4jTest1" bundle's importing of package "org.apache.log4j". Then the program would run correctly, and Clap would not find any runtime type error or loading constraint violation.

C. Analyzing JOnAS 5.2.0

We used Clap to analyze JOnAS 5.2.0's 77 active bundles on the same PC server as Section V.B, the analysis finished after about 19 minutes, and no runtime type error nor loading constraint violation was detected.

Generally, no warning does not mean no problem. Many Java dynamic features such as calling method by reflection

```
org.apache.log4j.Logger logger =
        org.apache.log4j.Logger.getLogger(X.class);
org.apache.log4j.net.SMTPAppender appender = new
        org.apache.log4j.net.SMTPAppender();
// The code for initializing appender is omitted
logger.addAppender(appender);
logger.error("Hello World");
```

Figure 9. Some code in "Log4jTest1" bundle

and the use of custom class loaders (not those provided by component containers) inside components render our approach neither sound nor complete.

D. About the Results of Case Studies

In some of our case studies, some propagation traces given by Clap are fairly long, amounting to hundreds of edges. When the trace is longer, the manual reviewing of this warning is more difficult and the chance that this warning is a false positive is higher. Some works [16, 17] combine static analysis and testing, and can be used to some extent to check for false positives; these approaches complement our work.

A variable with possibly wrong-typed pointed-to allocation sites may cause several warnings of runtime type errors. For example, if we copy line 17 in Fig. 3 several times, then all these lines will be detected, although all the warnings are about the same variable *baseVar*. This also makes manual reviewing more laborious.

VI. Related Work

Static analysis is used for detecting unsafe casts in Java programs [5-8], and unsafe casts are a source of runtime type errors. These proposed approaches, either use points-to analysis or use constraint-based analysis, only considers static types, so they cannot detect runtime type errors caused by class loaders.

Sawin and Rountev [18] propose a semi-static approach to improve the static resolution of dynamic class loading, using dynamically retrieved values of environment variables. This work tries to statically determine the value of parameter (class name) to "loadClass" method, but does not consider the behavior of class loaders so the runtime types of loaded classes cannot be obtained. Bodden, et al. make TamiFlex [19] to gain a better resolution of reflection calls and dynamic class loadings, using information gathered from recorded program runs. Our work have not considered explicitly invoking class loaders to load classes in program code yet, and their work may complement ours.

Some researchers formalize Java class loading [20-22], mainly proposing formal specifications of type-safety criteria for JVM. Our work is also about type-safety. In fact, some of these thoughts are incorporated into modern JVM as part of class loading and bytecode verification scheme, and thus has become an indirect cause of runtime type errors in component-based Java programs. However, these reasoning-based approaches are impractical for verifying type-safety of industry-scale Java programs.

Partial evaluation [23], which is a technique for program transformation and specialization, builds on the insight that

some dynamic program constructs have statically known behavior, which is somewhat similar to ours. Braux and Noyé use partial evaluation to tackle Java reflection [24]. But we have not seen work which partially evaluates Java class loaders so far.

Applications which use a lot of frameworks and third-party libraries are called *framework-intensive applications* in [25], and this work describes author's research plan to apply *blended analysis* technique to taint analysis. [26] and [27], which are also the author's works, mainly use blended analysis to detect performance problems in framework-intensive applications. The programs we focus on are largely also framework-intensive applications, but these works do not make use of the behavior of class loaders and thus cannot detect many of the runtime type errors.

VII. Conclusion and Future Work

In this paper we propose an approach using points-to analysis and dynamically gathered behavior information of Java class loaders to statically detect runtime type errors in component-based Java programs, and implement a prototype tool for OSGi.

In the future, we plan to make Clap couples looser with component container such as Felix, using mainly AOP code to interact with system under analysis, and then we will be able to easily create new versions of Clap for various component containers like web application servers. We will try to integrate test-based approaches [16, 17] to check if a warning is a false positive.

We are trying to conduct static analysis on component-based Java programs. Component containers often use reflection, dependency injection and other techniques which pose difficulties for static analysis. Still, some behavior of component containers can be statically simulated, given the configuration files, in order to cope with some of these difficulties. In the future we will try to push this idea further.

Acknowledgment

This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2009CB320704, the National Science and Technology Major Project of China under Grant No. 2011ZX03002-002-01, the National Natural Science Foundation of China under Grant No.61173004

References

[1] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*: Addison-Wesley, 2005.

[2] I. Sommerville, *Software Engineering*: Addison-Wesley Publishing Company, 2007.

[3] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] "Osgi service platform, core specification, release 4, version 4.1," OSG Alliance, 2007.

[5] O. Lhoták and L. Hendren, "Context-Sensitive Points-to Analysis: Is It Worth It?," in *Compiler Construction*. vol. 3923, A. Mycroft and A. Zeller, Eds., ed: Springer Berlin / Heidelberg, 2006, pp. 47-64.

[6] M. Sridharan and R. Bodik, "Refinement-based context-sensitive points-to analysis for Java," presented at the Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, Ottawa, Ontario, Canada, 2006.

[7] T. Wang and S. F. Smith, "Precise Constraint-Based Type Inference for Java," presented at the Proceedings of the 15th European Conference on Object-Oriented Programming, 2001.

[8] O. Lhoták, "Program analysis using binary decision diagrams," School of Computer Science, McGill University, Montreal, 2006.

[9] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using SPARK," presented at the Proceedings of the 12th international conference on Compiler construction, Warsaw, Poland, 2003.

[10] K. Gama and D. Donsez, "A Self-healing Component Sandbox for Untrustworthy Third Party Code Execution," in *Component-Based Software Engineering*. vol. 6092, L. Grunske, R. Reussner, and F. Plasil, Eds., ed: Springer Berlin / Heidelberg, 2010, pp. 130-149.

[11] L. O. Andersen, "Program analysis and specialization for the C programming language," PhD, Computer Science Department, University of Copenhagen, 1994.

[12] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Trans. Softw. Eng. Methodol.,* vol. 14, pp. 1-41, 2005.

[13] B. Bellamy, P. Avgustinov, O. d. Moor, and D. Sereni, "Efficient local type inference," presented at the Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, Nashville, TN, USA, 2008.

[14] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," presented at the Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, Mississauga, Ontario, Canada, 1999.

[15] R. Vallée-Rai and L. J. Hendren, "Jimple: Simplifying Java Bytecode for Analyses and Transformations," Sable Research Group, McGill University, 1998.

[16] C. Csallner and Y. Smaragdakis, "Check 'n' crash: combining static checking and testing," presented at the Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, 2005.

[17] Z.-Q. CUI, L.-Z. WANG, and X.-D. LI, "Target-Directed Concolic Testing," *Chinese Journal of Computers,* vol. 34, pp. 953-964, 2011 (in Chinese with English abstract).

[18] J. Sawin and A. Rountev, "Improving static resolution of dynamic class loading in Java using dynamically gathered environment information," *Automated Software Engineering,* vol. 16, pp. 357-381, 2009.

[19] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," presented at the Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, 2011.

[20] A. Tozawa and M. Hagiya, "Formalization and Analysis of Class Loading in Java," *Higher-Order and Symbolic Computation,* vol. 15, pp. 7-55, 2002.

[21] Z. Qian, A. Goldberg, and A. Coglio, "A formal specification of Java class loading," presented at the Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Minneapolis, Minnesota, United States, 2000.

[22] T.-j. Zuo, J.-g. Han, and P. Chen, "Formalizing Java Dynamic Loading in HOL," in *Theorem Proving in Higher Order Logics*. vol. 3223, K. Slind, A. Bunker, and G. Gopalakrishnan, Eds., ed: Springer Berlin / Heidelberg, 2004, pp. 79-90.

[23] N. D. Jones, "An introduction to partial evaluation," *ACM Comput. Surv.,* vol. 28, pp. 480-503, 1996.

[24] M. Braux and J. Noyé, "Towards partially evaluating reflection in Java," presented at the Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, Boston, Massachusetts, United States, 1999.

[25] B. Dufour, "Blended analysis for improving the quality of framework-intensive applications," presented at the 2008 Foundations of Software Engineering Doctoral Symposium, Atlanta, Georgia, 2008.

[26] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," presented at the 2007 international symposium on Software testing and analysis, London, United Kingdom, 2007.

[27] B. Dufour, B. G. Ryder, and G. Sevitsky, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," presented at the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Atlanta, Georgia, 2008.