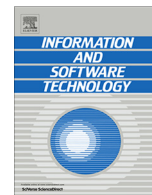
Information and Software Technology 56 (2014) 1076–1085

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



A class loading sensitive approach to detection of runtime type errors in component-based Java programs

Wenbo Zhang [a,*], Xiaowei Zhou [a,b], Jianhua Zhang [a,b], Zhenyu Zhang [a,c], Hua Zhong [a,c]

[a] Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
[b] Graduate University, Chinese Academy of Sciences, Beijing 100190, China
[c] State Key Laboratory of Computer Science, Beijing 100190, China

ARTICLE INFO

Article history:
Received 25 June 2013
Received in revised form 20 February 2014
Accepted 2 April 2014
Available online 26 April 2014

Keywords:
Runtime error detection
Class loading
Component-based

ABSTRACT

Context: The employment of class loaders in component-based Java programs may introduce runtime type errors, which may happen at any statement related to class loading, and may be wrapped into various types of exceptions raised by JVM. Traditional static analysis approaches are inefficient to detect them.
Objective: Our previous work proposed a semi-static detection work based on points-to analysis to detect such runtime type errors. In this paper, we extend previous work by referencing the information obtained from class loading to detect runtime type errors in component-based Java programs, without the need to running them.
Method: Our approach extends the typical points-to analysis by gathering the behavior information of Java class loaders and figuring out the defining class loader of the allocation sites. By doing that, we obtain the runtime types of objects a reference variable may point to, and make use of such information to facilitate runtime type error detecting.
Results: Results on four case studies show that our approach is feasible, can effectively detect runtime errors missed by traditional static checking methods, and performs acceptably in both false negative test and scalability test.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

With the increasing adoption of component-based software development (CBSD) as a mainstream approach of software engineering [23], Java programs have been the most prevalent software in Web world, such as Servlet/JSP, EJBs, OSGi[1]-based programs, and so on. Error detections of Java programs are invaluable for their comprehensive application, while type checking is one of such detection mechanisms. Type checking for Java can be done statically or dynamically. Type-related defects missed by Java compilers and captured by JVM's runtime type checking are conventionally named runtime type errors [12]. Such errors are very common in Java program, such those caused by unsafe casts [18].

Runtime type errors are usually apt to occur in component-based Java programs for the following reasons. First, in component-based Java programs, component containers, like web application servers and OSGi frameworks, different custom class loaders are often allowed, and classes are defined at runtime [14]. For example, in OSGi-based programs, classes in each bundle (OSGi-compliant component) are defined by the bundle's class loader [21]. Nevertheless, in Java web systems, classes of the web application and those of its hosting application server are also defined by different class loaders. The inconsistent of class loaders in classes loading contributes in the majority of runtime errors in Java. In this paper, we will focus on these scenarios and give our solution to detect this kind of runtime errors.

Second, it is very common 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 extensive use of application frameworks and third-party libraries. The instance of a class or its subclass created in one component may propagate to other

* Corresponding author. Tel.: +86 10 62661583 630.
E-mail address: zhangwenbo@otcaix.iscas.ac.cn (W. Zhang).
[1] http://www.osgi.org.

[2] Also known as *duplicated classes*.
[3] When we mention *class name*, we mean its fully qualified name, which includes the name of the Java package containing the class.

http://dx.doi.org/10.1016/j.infsof.2014.04.005
0950-5849/© 2014 Elsevier B.V. All rights reserved.

components that contain a class of the same name. Since same-named classes in different components are defined by different class loaders and thus represent different runtime types [12], the propagation may cause some reference variables to point to objects with wrong runtime types. Compilers, which focus more on static types, cannot detect this kind of. In realities, such errors are mostly found by JVM as runtime errors and handled as exceptions, such as *ClassCastException* and *ArrayStoreException*.

Eliminating same-named classes will prevent this kind of errors. A brute force solution is to delete those classes until there is only one class left for each class name. However, this approach may have undesired results. Same-named classes may have separate implementations, because they may come from different third-party libraries, and each of them may have a set of static fields, which take effect when loaded. If only one such class 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 will become the same-named classes are usually not known until those components are integrated together, especially when many third-party libraries are integrated at the same time. Second, components may come from various vendors and constraint these vendors comply with the same coding standard is also ineffective. Third, some third-party components are casually migrated from legacy code, such as the official OSGi-compliant log4j 1.2.16, which *imports* and *exports* [21] many Java packages and the clarity of component interface is sacrificed. Exceptions caused by runtime type errors may occur at almost every possible position of the program, rendering writing exception handling code to recover from these errors can be very hard [18].

Statically detecting runtime type errors will help programmers find out faults at early stage and enables corresponding remedies. There have been some works using static analysis to detect runtime type errors caused by unsafe casts [16,17,24,30]. However, these works do not consider runtime type discrepancies caused by class loaders and thus cannot detect runtime type errors effectively.

In our previous work [33], we propose a class loading sensitive 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 figure out the defining class loader of the allocation sites [15]. Then the runtime types of objects a reference variable may point to can be obtained. In this paper, we extend our previous work to acquire the runtime types of the reference variables from the behavior information of class loaders. Based on these runtime types, we check every program statement where JVM may raise exception [3] for runtime type error, and assess the possibility that related variables pointing to wrong-typed allocation sites. Besides, we also give the formal descriptive pseudo code to integrate our detection progress based on both points-to analysis and class loader information.

We implement our method as a prototype tool and conduct four case studies to show the feasibility and effectiveness of our method and its performance in false negative test and scalability test.

Contribution of this work is at least three-folded. First, we give a solution to detect runtime type error related to class loaders. Second, we use the de facto dynamic module system OSGi [11] as the framework to implement our open-source prototype tool. Third, we conduct a case study to validate our method and show it promising.

The remainder of this paper is organized as follows. Section 2 gives preliminaries by stating the problem of runtime type errors in component-based Java programs. Section 3 gives a short motivation and presents our approach in Section 4. Section 5 talks about the implementation of our prototype tool as an evaluation, presents results of case studies, and talks about threats to validity of the results observed, followed by Section 6, which introduces related work. Section 7 concludes the paper and gives future work.

2. Preliminaries

Before we give the problem and elaborate on our solution, we first introduce the class loading mechanism and OSGi framework as preliminaries.

2.1. Class loading in Java

In Java, all the classes are loaded into JVM by class loaders [14] 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 initiating class loader of the class, 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 same. A runtime class is identified both 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 in them. As a result, 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] [14] 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.

2.2. Type error detection

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 [16].

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 along the execution trace and many objects may be created but of the same type. Thus, the abstraction of objects to allocation sites will satisfy the need of checking for type errors.

2.3. The OSGi framework

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

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

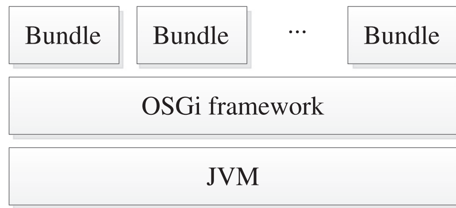


Fig. 1. OSGi-based program.

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 metadata 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.

3. Motivation

In this section, we use an example to motivate our work and show that class loader can be statically referenced as clues to detect runtime type errors.

3.1. An example

We use a simplest working example to show runtime type errors [5] in an OSGi-based program. In this example, "Test1" and "Test2" are two bundle, shown in Fig. 2. Both of them contain the class named "base.Base" where "base" is the package name. Further, "Test2" contains another class "test2.Derived" which is a subclass of the class "base.Base". The bundle "Test1" will invoke the class "test2.Derived", and we add "Import-Package: test2" to the meta data (MANIFEST.MF file) of "Test1", and "Export-Package: test2;uses:="base"" to that meta data of "Test2". Some codes of the "Test1" is also shown in Fig. 3.

3.2. The problem

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. 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 they interact with each other, some variables may be assigned wrong-typed object references, and runtime type errors may occur.

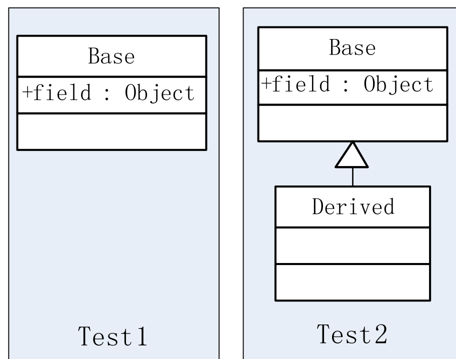


Fig. 2. Example bundles "Test1" and "Test2".

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

Fig. 3. Some code in Test1 bundle.

In the example, the two bundles can be successfully compiled but will get a *java.lang.VerifyError* when running. This is because that the variable *baseVar* points to a wrong-typed object when being used in field reference "baseVar.field" in Fig. 2. At runtime, "test2.Derived" is a 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. As a result, a runtime type error occurs.

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 those caused by unsafe casts. However, for some component-based Java programs, the causes of runtime time type errors are more complicated. At runtime, classes in different components may be defined by various class loaders. And some components may contain same-named classes. It happens that some variables are assigned wrong-typed object references having correct static types, e.g. the example in Section 3.1. Runtime type is determined by its corresponding static type and the defining class loader of the class involved in that static type.

The class loading scheme of Java, especially that of OSGi, is not easy for junior programmers to master and use. For example, we realize the exception handling and multi-threading cases. When writing component-based programs, some Java programmers may casually believe that same-named classes represent the same type, causing runtime type errors more often to happen.

3.3. Our basic idea

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 3.1 compiles successfully. However, since the defining class loader is introduced, same-named classes in different components will not be always identical and some static subclass relationships will not be valid at runtime. This kind of runtime type errors is caused by the "dynamic part" (defining class loaders) of runtime types.

Runtime type errors partially depend 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. For example, by referencing the class loader information in Fig. 2, a programmer can easily distinguish the same-named classes. However, Java compiler or static analysis cannot detect runtime type errors caused by defining class loaders generally.

3.4. Challenges

In this motivation example, we have shown that the behavior of class loaders provided by component containers is useful in detecting runtime type error. However, we also foresee some challenges. For example, how to integrate the use of class loader information with points-to analysis mechanism in detecting runtime type errors is not trivial. Also, the OSGi specification specifies a class loading mechanism, and servlet container Tomcat also has a

description of its class loaders. A well-designed detecting process will be elaborated on in the next section.

4. Our detection process

Previous points-to analysis approaches solely consider static types and cannot effectively 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.

4.1. Problem settings and typical points-to analysis

We first follow Andersen's [1] points-to analysis for Java [19] to give the basic problem settings, and then present our detection process. Here, 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. A typical detection process gradually constructs a points-to graph. For example, a points-to graph [19] example is shown in Fig. 4. 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. The goal is to find sites having suspicious type error in the graph.

Statements in the program under analysis may cause the propagation of object references, which is depicted by the transition function f. Table 1 shows the transition functions for five common statements, in which 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; and the function *dispatch* determines which method will be actually called, given an allocation site pointed to by the target variable of a virtual call statement and a method signature. Most of these transition functions have intuitive names. 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. 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, $l \cdot f_1 = r \cdot f_2$ can be transformed to $v = r \cdot f_2$ followed by $l \cdot f_1 = v$, in which v is a temporary variable.

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

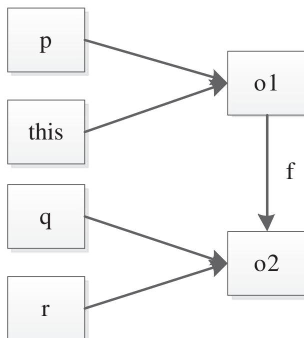


Fig. 4. Points-to graph (*p*, *this*, *q*, *r* are reference variables, *o*1, *o*2 are allocation sites, *f* is a field of *o*1).

Table 1
Transition functions of 'Anderson's style points-to analysis.

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

4.2. Referencing class loader information

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

4.2.1. 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).

In Section 5.5, we cite OSGi and JVM documents to show that the two assumptions almost hold.

4.2.2. Loading function

Under the above assumptions, we invoke the class loaders to get their behavior 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. Since the program is not deployed and run, our method is deemed static analysis.

We model the behavior of loading function as follows:

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

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 by null in JVM.

Note that in the 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.

4.2.3. Extending typical points-to analysis

We notice that same-named classes defined by different class loaders are different. Further, the reference variables declared in different runtime classes (fields and local variables) are also different. Therefore, we attach a runtime class C to every reference variable to differentiate between variables in same-named classes defined by different class loaders. We thus attach a defining class loader to every allocation site; with which we are able to obtain its runtime type, since the static type of an allocation site can be easily retrieved. After that, 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 that variable r in runtime class C may point to allocation site o_i with defining class loader CL. Here, 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

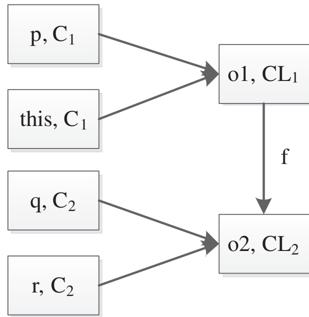


Fig. 5. Extended points-to graph.

defining class loader CL_2. An example of extended points-to graph using the same notations is shown in Fig. 5.

The extension of transition functions is shown in Table 2. We add an argument C to transition functions to denote the runtime class in which the argument statement resides. For the object creation case, the defining class loader of C will be used to find referred classes, according to JVM's resolution mechanism [3]. 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, and we make a special transition function f' to deal with it. Here, function *dispatch* uses runtime subclass relationship to find the actually invoked 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 the runtime super class of a class. We have modified the representation of points-to graph and transition functions. Note that the process of analysis is not affected.

With such extended points-to analysis, the runtime type of allocation sites pointed to by a reference variable can be retrieved.

4.3. Detecting runtime type errors

With the extended points-to analysis, we can check whether 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 3.1, if we delete line 17 in Fig. 3, 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 much attention to whether a local variable points to a wrong-typed object only at some points [2] (for example, when the variable is to be assigned to a field or returned by a method).

To solve the above problem, we statically simulate the bytecode verification mechanism of JVM by choosing some statements (or some fragment of statements) that 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 [27] to convert the program under analysis to Jimple intermediate code [28]. Jimple has only 15 kinds of statements, which makes the analyzer program simpler to implement. With such Jimple code representation, there are only six cases to check for runtime type errors.

4.3.1. 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 $\langle v, C\rangle{\cdot}type$ 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.

4.3.2. 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 $\langle v_1, C\rangle.type$.

4.3.3. 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.

4.3.4. Type casting ((T)v)

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

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

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

4.3.5. Method calling ($v_0.m(v_1, v_2, \ldots)$)

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:

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

Table 2
Extended transition functions.

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

4.3.6. Method returning (return v)

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

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

4.4. Complexity and other issues

For component-based Java programs, *C.defCL* in the transition function for the object creation cases is usually decided by the location of the class file. At the same time, the classes in a bundle are defined by its class loader, according to the specification of OSGi.

Our extension to points-to analysis increases the time and space complexity of the analysis. Since there are a lot of points-to analysis implementations and each implementation has its own complexity, we may not give a precise complexity formula. For an extended points-to graph, the reference variables in same-named classes may be distinguished by the defining class loaders of their belonging (containing)classes, and the number will not exceed the number of reference variables contained in these same-named classes. On the other hand, the number of allocation sites and fields is similar to each other. Further, suppose we change the names of all the same-named classes to make every class name different from each other. Our extended points-to analysis of original classes may not have a higher complexity than ordinary points-to analysis of these changed name classes.

For example, suppose there are 1200 classes with 1000 different names, ordinary points-to analysis may analyze 1000 classes; extended points-to analysis may not have higher complexity than analyzing 1200 classes using ordinary points-to analysis. So we think that the complexity of our method is generally acceptable.

5. Evaluation

To validate the feasibility of our method, we implement a prototype tool called Clap,[5] which is specific to OSGi-based programs now. This tool is based on Java static analysis framework Soot and OSGi framework Felix.[6] 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 is simulated by invoking the code of a slightly modified version of bundle class loader provided by Felix. We put the bundles under analysis into Felix's deployment folder, adjust Felix's configuration file (set boot delegations and so on), and start the analysis by just starting Clap. Clap starts the analysis after Felix installs and resolves every bundle, and none of the bundles starts and runs during the process. The system composed of Clap and the program under analysis is shown in Fig. 6.

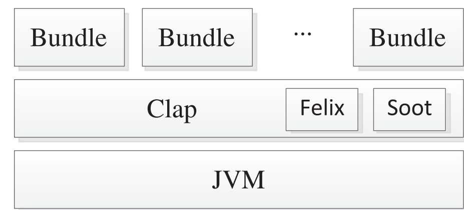


Fig. 6. Clap system structure.

5.1. Implementation details

5.1.1. Entry points

In Soot, points-to analysis should have entry points. For ordinary Java programs, the entry points contain the 'main' method. For OSGi-based programs, we choose the methods for activating and deactivating bundles as entry points.

5.1.2. Loading constraint violations

In OSGi-based programs, same-named classes may also cause JVM to raise *LinkageError* due to *loading constraint* violations. In Clap, we implement this checking which also uses defining class loaders of classes and loading function. The principle of this checking is a static simulation of JVM's process of enforcing loading constraints [14].

[5] The source code of Clap is available at <http://code.google.com/p/clap/>.

[6] http://felix.apache.org/site/index.html.

5.2. Pseudo code of checking algorithm

The pseudo-code description of our algorithm to check runtime type error is shown in Fig. 7.

From the pseudo-code, we have the following observations. For one line of Jimple statement *stmt*, if the statement is an InvokeStmt or contains (at most one) InvokeExpr, the number of calls of *checktype* function depends on the number of parameters of the method to invoke. Usually, a Java method does not have arbitrarily large number of parameters. We assume that the largest number of parameters of every method is k_1. For other kinds of statements, the algorithm code will call *checktype* function at most a certain number of times, say k_2. Here, we let $k = k_1 + k_2$, and the number of Jimple lines of the program under analysis is L, and then the number of calls of *checktype* function is not more than kL.

The running time of *reachingObjects* method depends on the implementation details of the points-to analysis. In some points-to analysis implementations, points-to set are computed when *reachingObjects* is called, whereas some others just return the pre-calculated points-to set. As a result, we attribute the total time of calling *reachingObjects* method to the time of performing points-to analysis. The execution time of the other code of *checktype* functions proportional to the size of the points-to set, if we do not count the time of handling detected runtime type errors.

5.3. Experiment design and settings

We use four programs to conduct case studies to understand the feasibility, effectiveness, false negative issue, and scalability of our method, respectively.

In the case study, we configure demand-driven context-sensitive points-to analysis as traversing at most 75,000 nodes per query, and set maximum refinement passes to 10 [24]. We set "ignore-types" to false and change code to let the points-to analysis consider static types only. It improves the performance by filtering out some obviously impossible propagation, but preserves statically well-typed ones to enable our analysis.

5.4. Case studies and results

5.4.1. Case of a toy program: feasibility test

We used Clap to analyze the example program in Section 3.1; the analysis finished after a couple of seconds on an ordinary PC (Core i7 3.4 GHz, 4 GB 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. The errors confirmed.

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 to have

```
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

      if stmt instanceof AssignStmt then
        l = stmt.getLeftOp()
        r = stmt.getRightOp()
        if r instanceof CastExpr then
          // type casting case
          castToType = r.getCastType()
          castOp = r.getOp()
          checktype(castOp, castToType)
        elseif r instanceof InstanceFieldRef then
          // instance field referring case (1)
          refBaseVar = r.getBase()
          baseType = refBaseVar.getType()
          checktype(refBaseVar, baseType)
        elseif r instanceof InvokeExpr then
          // method calling case (1)
          args = r.getArgs()
          argTypes = r.getMethodRef().parameterTypes()
          for each (arg, argType) in (args, argTypes)
            checktype(arg, argType)
          endfor
          if r instanceof InstanceInvokeExpr then
            invkBaseVar = r.getBase()
            baseType = invkBaseVar.getType()
            checktype(invkBaseVar, baseType)
          endif
        endif

        if l instanceof FieldRef then
          if r instanceof Local then
            refFieldType = l.getType()
            checktype(r, refFieldType)
          endif
          if l instanceof InstanceFieldRef then
            // instance field referring case (2)
            refBaseVar = l.getBase()
            baseType = refBaseVar.getType()
            checktype(refBaseVar, baseType)
          endif
        elseif l instanceof ArrayRef then
          // array storing case
          refArrBaseType = l.getType()
          checktype(r, refArrBaseType)
        endif
      elseif stmt instanceof InvokeStmt then
        // method calling case (2)
        // the same as AssignStmt and r is an InvokeExpr
      elseif stmt instanceof ReturnStmt then
        // method returning case
        retType = m.getReturnType()
        retOp = stmt.getOp()
        checktype(retOp, retType)
      endif
    endfor
  endfor
endfor

function checktype(localvar, intendedsupertype)
ptoSet = pointsto.reachingObjects(localvar)
  for alloc_site in ptoSet
    ptoType = alloc_site.getType()
    if ptoType is not a subtype of intendedsupertype then
      got runtime type error
    endif
  endfor
end function
```

Fig. 7. Checking algorithm.

malformed value. The last two edges do not have a corresponding source code line number. Let us look at the Jimple code of "testFieldRef" method to give a more clear view (Fig. 9, generated by our modified Soot). Our modified Soot considers defining class loader in inferring types of local variables [2]. 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 16in Fig. 3, and the last two edges are derived by this cast statement.

5.4.2. Case with more propagation: effectiveness test

We made an example about wrong-typed objects propagating via object fields. It is shown in Fig. 8.

Like Fig. 3, "FieldTest1" and "FieldTest2" are bundles. In "FieldTest2", there is a "BaseContainer" class, which is a singleton, holding a "Derived" instance in its Object-typed field "base". In "FieldTest1", some code retrieves the "BaseContainer" instance, gets the content of its "base" field and casts it to type "Base". This program can pass compilation but results in a *ClassCastException*-atruntime.

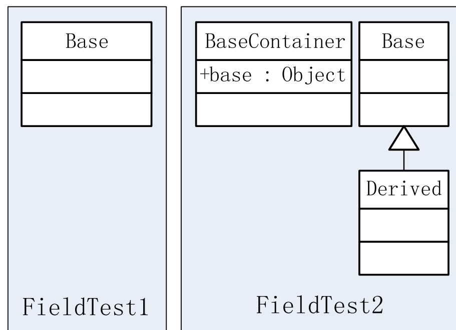


Fig. 8. Example about propagating via object fields.

```
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;
}
```

Fig. 9. The Jimple code of testFieldRef.

We used Clap to analyze this program in the same environment as Section 5.4.1) and the analysis finished in a couple of seconds. Clap gave out a runtime type error warning of case "type casting". This warning is also confirmed. The propagation trace contains 2 edges. The first one is an allocation edge indicating the creation of the "Derived" instance. The second one is a match edge which is a special kind of edge made by the demand-driven analysis. The match edge indicates the writing-reading pair of the "base" field of "BaseContainer" class.

5.4.3. Case in a bug-fix scenario: false negative test

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. 10, in which class"Logger" and "SMTPAppender" come from different bundles. In addition, we 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 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

```
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");
```

Fig. 10. Some code in "Log4jTest1" bundle.

"Log4jTest1" bundle may point to objects created in "Log4jBundle" bundle.

We used a simple approach to correct the problem by removing the importing of package "org.apache.log4j" in the bundle "Log4jTest1". After that, the program run correctly, and Clap reports no runtime type error or loading constraint violation any longer.

5.4.4. Case of a large program: scalability test

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

5.4.5. Summary

With the four designed case studies, we have the following observations.

- Our method is feasible.
- Our method works well in normal cases.
- Our method has the capability of realizing the fix of the bug and avoiding false negative.
- Our method can handle programs of medium scale.

5.5. Threats to validity and related discussions

Generally speaking, there is no guarantee of a bug-free program even if no warning is reported. Many Java dynamic features such as calling method by reflection and the use of custom class loaders (not those provided by component containers) inside components render our approach neither sound nor complete.

In some of our case studies, some propagation traces given by Clap are fairly long, e.g., hundreds of edges. When the trace is long, the manual reviewing of this warning is more difficult and the chance that this warning is a false positive is high. Some works [6,7] combine static analysis and testing, and can be used to some extent to check for false positives; these approaches complement the scope of this 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.

Further, in Section 4.2, we base our method on two assumptions. If a bundle's class loader initiates the loading process of a class, the overall process to search for the class is as follows.

1. If the class is in the "java.*" package, the loading request is delegated to parent/system class loader. If parent/system class loader cannot find the class, then the process fails (without continuing the following steps).
2. If the class's package name is directly or indirectly included in the boot delegation list (, which can be set by modifying the configuration of OSGi framework), the parent/system class loader is used to try to find the class. If the class cannot be found, the search continues.

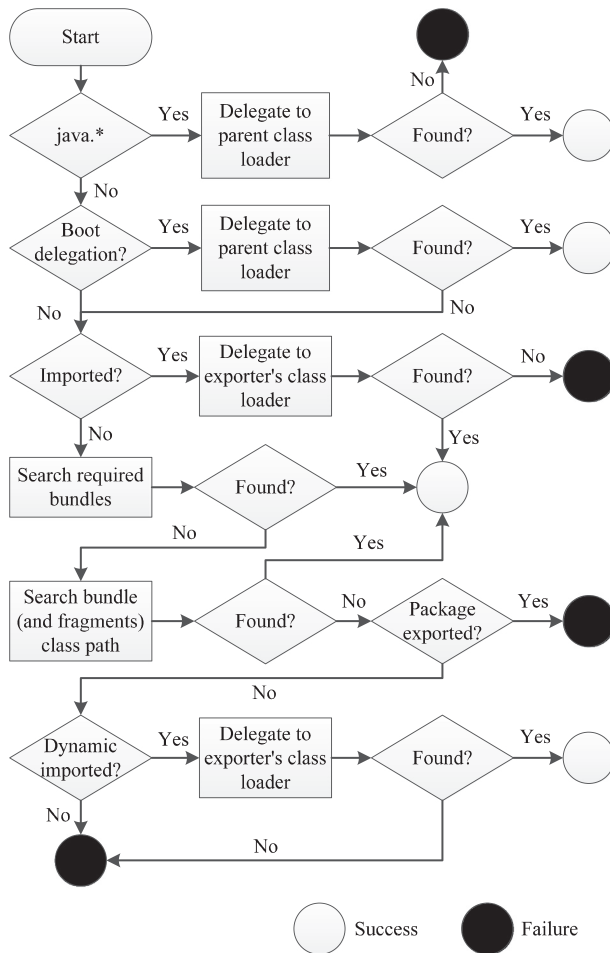


Fig. 11. The flow chart of bundle's class loader's search workflow.

3. If the package name is imported using the Import-Package header in the bundle's meta data, the request is delegated to the exporting bundle's class loader. If it is not found, the process fails.
4. If the package name is not imported using the Import-Package header, the bundles cited using the Require-Bundle header are searched for the class. If it is not found, the search continues.
5. The bundle's own internal bundle class path and the class path of the bundle's fragment bundlesare searched for the class. If it is not found, the search continues.
6. If the package name is directly or indirectly included using the DynamicImportPackage header, the dynamic importing of the package is tried. If the package is successfully imported, the request is delegated to the exporting bundle's class loader. If it is not found, the process fails.

These steps are connected in a workflow shown in Fig. 11 (This figure is a slightly simplified version of that in [21]). The procedure implements lots of favorable features of OSGi. For example, the codes in one bundle cannot use all the classes in other bundles, because the class paths of the other bundles are not searched unless they are imported. This feature shows information hiding and explicit interface characterization of CBSD. Every OSGi framework implementations, like as Apache Felix, implement such a workflow. And this workflow is proved to be terminable; moreover, after all the bundles are resolved (every mandatory import of every bundles wired to a corresponding exporter), different executions of the workflow at the same starting state will have the same result.

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. However, we also realize that there may be scenarios where the two assumptions do not hold.

At last, our method is built based on the Andersen's settings. Since we propose a general method in this work, there is no limitation on the points-to analysis mechanism chosen. Applying our method on other implementation of points-to analysis method may result in different observations in the case studies.

6. Related work

Static analysis is used for detecting unsafe casts in Java programs [16,17,24,30], and unsafe casts are a source of runtime type errors. The proposed approaches use either points-to analysis or constraint-based analysis. They only consider static types and cannot detect runtime type errors caused by class loaders.

Points-to analysis is used to statically estimate which objects a reference variable may point to during execution of the program under analysis. Spark [15] is an early but still popular context-insensitive points-to analysis framework for Java.

There is work showing that context-sensitive points-to analysis is more precise than context-insensitive one [16]. 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 2 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 [24], which is efficient when we only need to get points-to sets for a small number of variables.

Context-sensitive analysis is far more time and space consuming and there are a lot of works on trying to reduce its complexity. Whaley and Lam [29] and Lhoták [17] use binary decision diagram (BDD) to make context-sensitive points-to analysis more scalable. Xu and Rountev [31] employ equivalent context merging to reduce time and space consumption. Xiao and Zhang [32] incorporate geometric encoding in context-sensitive analysis; this approach substantially reduces encoding redundancy and achieves great performance improvement. Demand-driven points-to analysis for Java are mainly proposed in [25] and [24] which are context-insensitive and context-sensitive respectively.

Sawin and Rountev [22] 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 [4] 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 [20,26,34], mainly proposing formal specifications of type-safety criteria for JVM. Our work is also related to 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 [13], 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 [3]. 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 [8], and this work describes author's research plan to apply *blended analysis* technique to taint analysis. [9] and [10], 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.

7. 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. Case studies show our method feasible, effective, and scalable.

In the future, we will first use larger and more complicated experiment to evaluate our method. Further, we plan to make Clap couples loosely with component container such as Felix, using mainly AOP code to interact with system under analysis. Thus, 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 [6,7] 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.

Acknowledgments

This work is supported by the National Key Basic Research Program of China (Grant No. 2014CB340701) and the National Natural Science Foundation of China (Grant No. 61173004, 61379045).

References

[1] L.O. Andersen, Program Analysis and Specialization for the C Programming Language, PhD, Computer Science Department, University of Copenhagen, 1994.
[2] B. Bellamy, P. Avgustinov, O. d. Moor, D. Sereni, Efficient local type inference, in: Proceedings of the Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, Nashville, TN, USA, 2008.
[3] M. Braux, J. Noyé, Towards partially evaluating reflection in Java, in: Proceedings of the Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, Boston, Massachusetts, United States, 1999.
[4] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, M. Mezini, Taming reflection: aiding static analysis in the presence of reflection and custom class loaders, in: Proceedings of the Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, 2011.
[5] Class Loading and Types in Java. <https://access.redhat.com/site/documentation/enUS/JBoss_Enterprise_Application_Platform/4.3/html/Server_Configuration_Guide/JBoss_JMX_Implementation_Architecture-Class_Loading_and_Types_in_Java.html>.
[6] C. Csallner, Y. Smaragdakis, Check 'n' crash: combining static checking and testing, in: Proceedings of the Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, 2005.
[7] Z.Q. Cui, L.Z. Wang, X.D. Li, Target-directed concolic testing, Chin. J. Comput. 34 (2011) 953–964 (in Chinese with English abstract).
[8] B. Dufour, Blended analysis for improving the quality of framework-intensive applications, in: Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium, Atlanta, Georgia, 2008.
[9] B. Dufour, B.G. Ryder, G. Sevitsky, Blended analysis for performance understanding of framework-based applications, in: Proceedings of the 2007 International Symposium on Software Testing and Analysis, London, United Kingdom, 2007.
[10] B. Dufour, B.G. Ryder, G. Sevitsky, A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, Atlanta, Georgia, 2008.
[11] K. Gama, D. Donsez, A self-healing component sandbox for untrustworthy third party code execution, in: L. Grunske, R. Reussner, F. Plasil (Eds.), Component-Based Software Engineering, vol. 6092, Springer, Berlin/Heidelberg, 2010, pp. 130–149.
[12] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Addison-Wesley, 2005.
[13] N.D. Jones, An introduction to partial evaluation, ACM Comput. Surv. 28 (1996) 480–503.
[14] T. Lindholm, F. Yellin, Java Virtual Machine Specification, Addison-Wesley Longman Publishing Co., Inc., 1999.
[15] O. Lhoták, L. Hendren, Scaling Java points-to analysis using SPARK, in: Proceedings of the Proceedings of the 12th International Conference on Compiler Construction, Warsaw, Poland, 2003.
[16] O. Lhoták, L. Hendren, Context-sensitive points-to analysis: is it worth it?, in: A. Mycroft, A. Zeller (Eds.), Compiler Construction, vol. 3923, Springer, Berlin/Heidelberg, 2006, pp. 47–64.
[17] O. Lhoták, Program Analysis using Binary Decision Diagrams, School of Computer Science, McGill University, Montreal, 2006.
[18] S. Liang, G. Bracha, Dynamic class loading in the Java virtual machine, in: Proceedings of the Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, New York, NY, USA, 1998.
[19] A. Milanova, A. Rountev, B.G. Ryder, Parameterized object sensitivity for points-to analysis for Java, ACM Trans. Softw. Eng. Methodol. 14 (2005) 1–41.
[20] Z. Qian, A. Goldberg, A. Coglio, A formal specification of Java class loading, in: Proceedings of the Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota, United States, 2000.
[21] Osgi service platform, core specification, release 4, version 4.1, OSG Alliance, 2007.
[22] J. Sawin, A. Rountev, Improving static resolution of dynamic class loading in Java using dynamically gathered environment information, Autom. Softw. Eng. 16 (2009) 357–381.
[23] I. Sommerville, Software Engineering, Addison-Wesley Publishing Company, 2007.
[24] M. Sridharan, R. Bodik, Refinement-based context-sensitive points-to analysis for Java, in: Proceedings of the Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, 2006.
[25] M. Sridharan, D. Gopan, L. Shan, R. Bodík, Demand-driven points-to analysis for Java, in: Proceedings of the Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, San Diego, CA, USA, 2005.
[26] A. Tozawa, M. Hagiya, Formalization and analysis of class loading in Java, Higher-Order Symb. Comput. 15 (2002) 7–55.
[27] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan, Soot – a Java bytecode optimization framework, in: Proceedings of the Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, Mississauga, Ontario, Canada, 1999.
[28] R. Vallée-Rai, L.J. Hendren, Jimple: Simplifying Java Bytecode for Analyses and Transformations, Sable Research Group, McGill University, 1998.
[29] J. Whaley, M.S. Lam, Cloning-based context-sensitive pointer alias analysis using binary decision diagrams, in: Proceedings of the Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, Washington DC, USA, 2004.
[30] T. Wang, S.F. Smith, Precise constraint-based type inference for Java, in: Proceedings of the Proceedings of the 15th European Conference on Object-Oriented Programming, 2001.
[31] G. Xu, A. Rountev, Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis, in: Proceedings of the Proceedings of the 2008 International Symposium on Software Testing and Analysis, Seattle, WA, USA, 2008.
[32] X. Xiao, C. Zhang, Geometric encoding: forging the high performance context sensitive points-to analysis for Java, in: Proceedings of the Proceedings of the 2011 International Symposium on Software Testing and Analysis, Toronto, Ontario, Canada, 2011.
[33] X. Zhou, W. Zhang, J. Zhang, Semi-static detection of runtime type errors in component-based Java programs, in: Proceedings of the Proceedings of the 19th Asic-Pacific Software Engineering Conference, Hong Kong, 2012.
[34] T.J. Zuo, J.G. Han, P. Chen, Formalizing Java dynamic loading in HOL, in: K. Slind, A. Bunker, G. Gopalakrishnan (Eds.), Theorem Proving in Higher Order Logics, vol. 3223, Springer, Berlin/Heidelberg, 2004, pp. 79–90.