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.

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/

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*.