Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 81–98
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Reverse-engineering 1-n associations from Java bytecode using alias analysis

Yoohoon Kang *, Chanjin Park, Chisu Wu

School of Computer Science and Engineering, Seoul National University, South Korea

Received 27 April 2005; received in revised form 15 February 2006; accepted 22 February 2006
Available online 5 April 2006

Abstract

1-n associations are design language constructs that represent one-to-many structural invariants for objects. To implement 1-n associations, container classes, such as *Vector* in Java, are frequently used as programming language constructs. Many of the current CASE tools fail to reverse-engineer 1-n associations that have been implemented via containers because sophisticated analyses are required to infer the type of elements collected in containers. This paper presents a new approach to reverse-engineering 1-n associations from Java bytecode based on alias analysis. In our approach, 1-n associations are inferred by examining the abstract heap structure that is constructed by applying an alias analysis on inter-variable relationships extracted from assignments and method invocations of containers. Our approach handles container alias problem that has been neglected by previous techniques by approximating the relationships between containers and elements at the object level rather than analyzing only the bytecode. Our prototype implementation was used with a suite of well-known Java programs. Most of the 1-n associations were successfully reverse-engineered from hundreds of class files in less than 1 minute.

Keywords: Reverse-engineering; Class diagram; Program understanding; Alias analysis; Static program analysis; Type inference

1. Introduction

Reverse-engineered design models are considered useful in software development. They are useful in guiding maintainers to gain an understanding of the system of interest and in allowing comparisons to be made with existing design models for checking design code traceability or round tripping. Most CASE tools support reverse-engineering facilities, an indication of just how useful this technique has become.

One of the important challenges facing the object-oriented reverse-engineering community is the inference of structural invariants for the objects that capture the problem space [2]. With the rise of *Model-Driven-Architecture* [17], it is becoming essential to extract such design knowledge for later transformations into other implementation-specific models.

A current study [1] on the state-of-the-art of CASE tools emphasizes that many available CASE tools fail to infer "one-to-many" structural invariants. The relationship between a car and its wheels is a typical example of such an invariant. The inference is difficult because one-to-many structural invariants may not be directly reflected by the code. Due to the gap in translation that exists between design and implementation [5], sophisticated code-level analyses are required to identify helpful clues for inferring these invariants.

One-to-many structural invariants are directly represented by design language constructs such as 1-n associations. For example, as noted before, a 1-n association represents the invariants for a car and its wheels. Therefore, one-to-many structural invariants can definitely be inferred by reverse-engineering 1-n associations instead.

On the other hand, during development, 1-n associations can be implemented in many ways. Container classes

* Corresponding author. Tel.: +82 2 874 4165.
E-mail addresses: rmaker@selab.snu.ac.kr (Y. Kang), cjpark@selab.snu.ac.kr (C. Park), wuchisu@selab.snu.ac.kr (C. Wu).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.02.004

are one of the most frequently used methods. A container, such as *Vector* in Java, is simply an object that groups multiple elements into a single unit [22]. Then, 1-n associations can be implemented with containers and their elements. In the case of the car and wheels example, a car may be implemented by using a container that stores wheels as elements.

To reverse-engineer the 1-n associations implemented with containers, it is essential to infer the concrete type of the elements stored in the container. This is especially true since Java supports subtype polymorphism and the elements in the container have a top-level type of *java.lang.Object* [18]. Therefore, sophisticated code-level analyses are required to infer the concrete type of the elements.

Our method is applicable to legacy Java code that does not support parametric polymorphism and to code that is based on user-defined, not-templated containers. The old version of Java code (1.4 or lower) is the primary target of our method.

Reverse-engineering 1-n associations could provide a solid foundation for other research. Recovery of design language constructs (like 1-n associations) is considered an essential prerequisite of reverse-engineering precise UML class diagrams [6] and detecting applied design patterns [15,16]. On the other hand, structural invariants for problem domain objects that are represented by 1-n associations play a crucial role in transforming *Platform-Independent-Model* to *Platform-Specific-Model* in *Model-Driven-Architecture* research.

Several approaches have been proposed recently to deal with this problem [3–6]. The predominant technique is the examination of a container field and the method invocations of the field. For example, suppose that we have a class *Queue* with a *Vector* field and a method invocation on the field:

```
class Queue {
    Vector slots = new Vector();
    void addSlot(Slot slot) {
        slots.addElement(slot);
...
```

This would result in a correct 1-n association from *Queue* to *Slot* with a role name *slots* because *Queue* has the container (*Vector*) field named *slots* and the method *addElement(Slot)* is called on the field with a parameter of type *Slot*. However, this approach has a drawback; it does not take into account the "aliases" for the container field. Aliases are two variables referring to the same object in the heap [9] and are made by assigning one variable to another variable. Suppose that the *Queue* class above has another *Vector* field and several assignments have been added as follows:

```
class Queue {
    Vector slots = new Vector();
    Vector p;
    void addSlot(Slot slot) {
        p = slots;
        Vector q = p;
        q.addElement(slot);
...
```

This may not result in the correct 1-n associations because *q* is not a container field. However, variable *q* is the alias for both *p* and *slots*, so all three variables refer to the same object in the heap. Without considering the effect of variable *q*, it is impossible to produce the correct 1-n associations from only the container field *slots*.

To overcome this drawback, alias analysis can be used for a more accurate heap-dependence analysis. Since one-to-many structural invariants can be viewed as properties of a runtime heap structure [2], it is crucial to analyze aliases that refer to the same object if reverse-engineering techniques are to determine the correct 1-n associations. To assist this alias analysis, our approach produces two 1-n associations from *Queue* to *Slot* with role names *slots* and *p* for the above example.

This paper presents a new approach to reverse-engineering 1-n associations implemented via containers from Java bytecode based on alias analysis. 1-n associations are inferred by examining the abstract heap structure that is a static approximation of the runtime heap structure related to containers and their elements. An abstract heap structure is a graph whose nodes represent approximate objects that we call a "rep object" and whose edges represent "referencing relationships" among the rep objects. Aliases for a container variable or an element variable are interpreted as a whole into a rep object. Inserting or extracting the element to or from the container is translated into a referencing relationship between the rep objects referred to by the container and the element.

Our approach handles the container alias problem that has been neglected by previous approaches by employing a novel application of alias analysis. By considering the effect of aliases for container variables, our approach can produce more correct 1-n associations.

In addition to the container alias problem, our approach handles other interesting problems related to iterators and nested containers [3,4]. Iterators that scan the collection of elements in a container are also analyzed to infer the element types of the container. Furthermore, nested containers are naturally expressed in the abstract heap structure.

The rest of this paper is organized as follows: Section 2 gives a detailed explanation of the problem that we set out to solve. Section 3 shows how the types of the elements stored in a container are inferred using alias analysis. Section 4 describes our reverse-engineering mechanism in terms of Java bytecode. Section 5 gives the experimental results that we obtained by applying our approach to a suite of Java programs. Section 6 discusses the heuristics used in our approach and the challenges that remain. Section 7 describes some related work. Section 8 gives our conclusions and recommendations for future research.