

Toward Comprehension of Side Effects in Framework Applications as Feature Interactions

Izuru Kume
*Graduate School of Information Science
 Nara Institute of Science and Technology
 Ikoma, Nara, Japan
 Email: kume@is.naist.jp*

Masahide Nakamura
*Graduate School of System Informatics
 Kobe University
 Nada, Kobe, Japan
 Email: masa-n@cs.kobe-u.ac.jp*

Etsuya Shibayama
*Information Technology Center
 The University of Tokyo
 Bunkyo, Tokyo, Japan
 Email: etsuya@ecc.u-tokyo.ac.jp*

Abstract—Application frameworks are widely used in order to increase efficiency and reliability in object-oriented software development. In this paper we put a focus on side effects caused by misuse of frameworks. A processes of such a side effect often includes cross-border method invocations between an application and its framework, and is difficult to resolve.

This paper proposes an approach to visualizing such a side effect as a feature interaction between a framework and its application. This paper shows a case study to apply our approach to a practical framework application, and discuss its practical usefulness.

Keywords—application frameworks; feature interactions; side effects; program comprehension;

I. INTRODUCTION

An application framework is a reusable software product which provides reusable design and implementation common to applications of a particular domain [1]. In recent years, application frameworks are used in various fields of software projects.

In this paper, we call classes in a framework *framework classes*. We also use the term *application specific classes* to refer to classes that implement application specific features within a framework application. We call a method defined in a framework class a *framework method*, while a method defined in an application specific class an *application method*.

A framework specifies *hot spots*, which represent application specific features and are implemented by application methods. Application methods implementing hot spots are invoked inside of the framework. This invocation style is called *inversion of control* [2]. On the other hand, frameworks often expose some methods so that applications can set states of the frameworks. Such state setting methods tend to introduce *deviant code* [3], which invokes exposed framework methods in a wrong way, and causes an erroneous state. We call such invocation style that an application method invokes a framework method, *re-inversion of control*.

This paper puts a focus on *deviant effects*. A deviant effect is a side effect which is triggered by re-inversion of control and brings an erroneous state inside of a framework.

A deviant effect by definition is caused by deviant code which misuses one or more of exposed framework methods. Monperrus et al.[3] point out that fixing deviant code requires program understanding. However, understanding deviant effects is difficult because of the complexity of frameworks.

To tackle with this problem, we propose an approach to visualizing a deviant effect as a kind of feature interaction [4], [5] between an application and its framework with support by dynamic analysis. Our visualization concisely illustrates where and how a feature interaction is triggered. Such an illustration is expected to be useful for fixing deviant code which is responsible for the feature interaction.

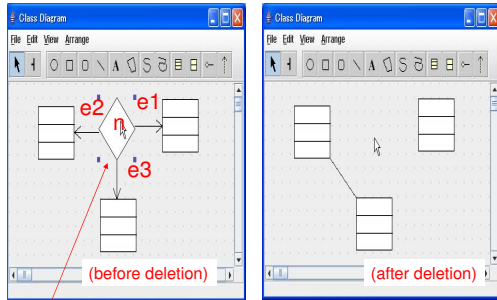
In the rest of this paper, We explain related work in section II. We introduce a motivating example of deviant effect in section III. We explain our approach in section IV, and discuss the result of applying our method to the deviant effect example in section V, and conclude our work in section VI.

II. RELATED WORK

A feature interaction is a situation where two or more features, each of which works correctly by itself, shows an unexpected result in their combination [4], [5]. Nakamura et al. [4] introduces an object-oriented model, called *environment interaction*, which describes feature interactions between services by digital appliances in a home network system. Their environment interaction represents the environment in a home network system as *the environment object*. A feature interaction occurs when a service sets a state of an environment object and another service references the state in its execution.

Side effects in object-oriented programs can be represented in a similar way to environment interaction. However, an aid by dynamic analysis is necessary for obtaining understandable results because of the complexity of practical object-oriented programs.

Dynamic object flow analysis by Lienhard [6] proposes a method to visualize dependency among classes and groups of classes. Lienhard's method is based on a data flow



Deleting the node of the 3-ary Association causes an exception.

```
java.lang.IndexOutOfBoundsException: Index: 1, Size: 0
at java.util.ArrayList.RangeCheck(ArrayList.java:547)
at java.util.ArrayList.get(ArrayList.java:322)
at org.tigris.gef.presentation.FigNode.dispose(FigNode.java:239)
at org.tigris.gefdemo.uml.ui.ModelNodeFig.dispose(ModelNodeFig.java:90)
at org.tigris.gef.base.SelectionManager.deleteFromModel(SelectionManager.java:769)
at org.tigris.gef.base.CmdDeleteFromModel.dolt(CmdDeleteFromModel.java:50)
at org.tigris.gef.base.Cmd.actionPerformed(Cmd.java:177)
```

Figure 1. Exception at Deleting Association Node

analysis which deals with various kinds of object references including object referenced by instance variables. However, her data flow analysis does not deal with (1) object references as method receivers, (2) actions on instance variables to assign and reference primitive values, and (3) control dependencies. Such kinds of information are necessary for analysis of side effects.

Wang et al. [7] proposed a practical means of generating compact program traces for dynamic slicing including data dependency via assignments and references of instance variables. Our approach assumes interactive trace analysis, which require richer kinds of information in program traces than theirs. We have developed a Java byte code instrumentation tool [8] which generates program traces with necessary kinds of information for our purpose.

III. MOTIVATING EXAMPLE

In the following we introduce a side effect example found in a framework application¹ built on a practical framework GEF² developed by a third party.

The framework and the application are written in Java. The application implements simple features to edit UML class diagrams. Figure 1 shows an exception thrown at deleting a 3-ary UML association.

The association consists of a node (n) and three edges (e1, e2, and e3). The association should be deleted by deleting its node (n), but actually we encounter an erroneous situation shown in figure 1. Note the error message in figure 1. This message suggests a deviant effect on an ArrayList instance, where an exception is thrown by an

¹<http://gefdemo.tigris.org/>

²<http://gef.tigris.org/>

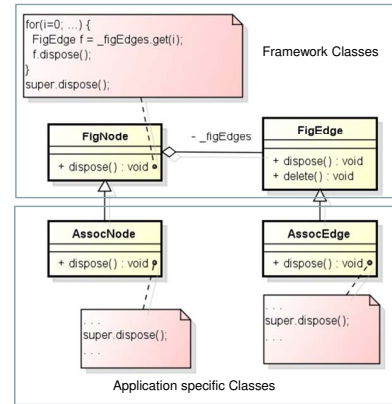


Figure 2. Class Diagram of Example Application

attempt to get an item from this instance which has already been empty.

Figure 2 shows a class diagram of this application. Framework classes FigNode and FigEdge represent graph nodes and graph edges, respectively. Application specific classes³ AssocNode and AssocEdge implement nodes and edges of UML n-ary associations by extending FigNode and FigEdge, respectively.

The framework specifies dispose() as a hot spot which represents a disposal process of a graph element. Each of the above framework classes implements dispose() to perform an internal disposal process. The application specific classes override dispose() to implement application specific disposal features. The overriding application methods invoke the overridden framework methods by super.dispose() in order to execute the framework disposal procedures. (See the pseudo code in figure 2.) As a result, a disposal process of an association node or an association edge includes nested inversion/re-inversion of controls.

IV. UNDERSTANDING DEVIANT CLASSES EFFECTS AS FEATURE INTERACTIONS

Maintainers need a support in detecting an application method which unexpectedly triggers a deviant effect resulting in a feature interaction. Such detection is the first step to detecting the deviant code which is responsible for the deviant effect, we believe. For this purpose, our approach aims at helping maintainers guess the trigger of a deviant effect, and at concisely visualizing the internal behavior of the guessed method in terms of object references.

We assume that maintainers understand behaviors of exposed framework methods in an abstract fashion. In our deviant effect case, we know that method FigNode.dispose() disposes a graph node and its

³We abbreviate the original class names for simplicity.

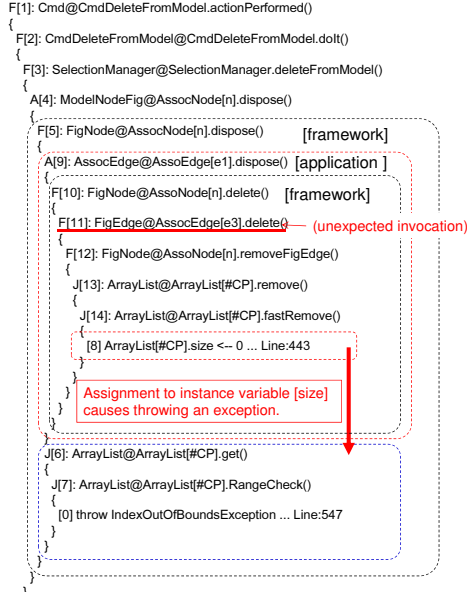


Figure 3. Invocation Tree

edges attached to the node, for example. We also assume a supporting tool for interactive program trace analysis. In this paper, we rely on a prototype tool built on our byte code instrumentation library [8]. Our prototype tool enables query on actions in a program trace, and dependency analysis among actions.

We introduce two kinds of visualization of a deviant effect. The first kind of visualization illustrates a method invocation tree participating in a deviant effect so that maintainers can find unexpected re-inversion of control which triggers the deviant effect. The second kind of visualization concisely illustrates how methods in the re-inversion triggers this deviant effect as a feature interaction. Figure 3 and figure 4 illustrate an invocation tree of the deviant effect example in section III, and the triggering process of a resulting feature interaction, respectively.

A shared object, which plays a similar role to environment object in an environment interaction [4], becomes the key to our visualization. We call such a shared object in a deviant effect, *the coupling point* of this deviant effect. Maintainers are responsible for specifying the coupling point of a deviant effect. Maintainers should also specify an action which shows an erroneous state brought by this deviant effect. In our example, the `ArrayList` instance is specified as the coupling point, and the exception throwing action is also specified.

The invocation tree in figure 3 is derived from the coupling point and the exception throwing action. The coupling point, which is the `ArrayList` instance, is denoted by `#CP` in this tree. The exception throwing action is assigned

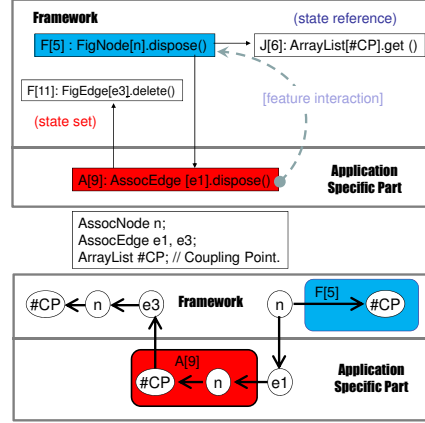


Figure 4. Feature Interaction

a unique ID [0]. By dependency analysis, we obtain action [8] which set a state of the coupling point, which causes the exception. The tree is obtained as the minimum invocation tree under which actions 0 and 8 are executed.

Each invocation expression has a signature with a unique ID number such as `A[9]`. It also has the name of the class that defines the method, the method receiver with its class name, and the method name. Signatures `A`, `F`, and `J` denote an application method, a framework method, and a Java library method, respectively. These signatures are necessary to grasp inversion/re-inversion of controls in an invocation tree.

Action [8] denotes assignment of `size`, which is an instance variable of the `ArrayList` instance. The assigned value 0 indicates that the `ArrayList` instance becomes empty, which brings the exception throwing action [0]. In this way, this figure illustrates the cause and the result of the deviant effect in the invocation tree. This cause-result relationship between actions [8] and [0] can be obtained by dependency analysis in the program trace.

Then, let's examine the re-inversion of control under which the above assignment action is executed. The association node (`node`) and its edges (`e1` and `e3`) in figure 1 appear as method receivers. Note that method `FigEdge.delete()` (`F[11]`) is invoked on `e3`, which is a sibling of `e1`, under `AssocEdge.dispose()` invoked on edge `e1` (`A[9]`). We did not expect that disposal of `e1` is accompanied by deletion of `e3`. We can guess that application method `A[9]` might trigger the deviant effect.

Figure 4 puts focuses on the suspicious re-inversion of control, and the invocation context where the exception is thrown. It illustrates cross-border method invocations between the application specific part and the framework, and cross border sequences of object reference via instance variables which bring the coupling point for setting and ref-

erencing its state. This result is derived from the specifying method invocations `F[5]`, `A[9]` and the coupling point by a partially manual visualization process.

This figure concisely illustrates at its upper side that application method invocation `A[9]` is the trigger of the deviant effect, whose result manifests itself in framework method `F[5]`. We can see that method `F[5]` fails because it invokes `A[9]`. Method invocation `F[11]` and `J[6]` are shown to indicate where the state of the coupling point has been set and the resulting exception is thrown, respectively.

At the lower side of figure 4, sequences of instance variable references to obtain the coupling point (`#CP`) are shown in an abstract fashion. The reference actions in `A[9]` and `F[5]` are surrounded by rounded rectangles. We can see that in method `A[9]` `edge e1` obtains `e3` through its node (`n`) to invokes `FigEdge.delete()`. It shows how `A[9]` triggers the feature interaction. We must examine the source code why this method does it.

By examining the source code, it is known that method `AssocEdge.dispose()` removes sibling edges of its receiver when just two siblings are left by disposal of its receiver. It is for replacing the current presentation of an n-ary association to that of a binary association. In the case of section III, the replacement occurred at the disposal of `e1`

Note that method `AssocEdge.dispose()` works well as long as it is not invoked in `FigNode.dispose()`. Therefore, we decided not to change the implementation of `AssocEdge.dispose()` but that of `A[5]` `ModelNodeFig.dispose()` which invokes `F[5]`. We changed the implementation so that all edges are deleted before invocation of `F[5]`. This new implementation prevents the problematic combination of `F[5]` and `A[9]`, and thus works well.

V. DISCUSSION

The illustration in figure 4 concisely shows that `AssocNode.dispose()` triggers the feature interaction by invoking `delete()` on `e3` which has been obtained through `e1` and `n`. In other words, this illustration explains 'where and how' the feature interaction has been formed. We had to examine the source code for further understanding: (1) why `AssocNode.dispose()` did it, and (2) where the deviant code, which is responsible for this feature interaction, is implemented. For understanding such details without source code examination, we need to enhance our behavioral model of invoked methods and our supporting dynamic analysis technique.

At this time, we need a partially manual process in visualizing a deviant effect due to a limitation of the current implementation. Our prototype tool also lacks implementation for maintainers to select objects and methods in section IV interactively. In future we will implement GUIs

to examine a program trace interactively, and to create a diagram to visualize its result automatically.

VI. CONCLUSION

In this paper we proposed an approach to visualizing a kind of side effect in a framework application, which we call *a deviant effect*, as a feature interaction. We applied our approach to a deviant effect in a practical framework application, and found where and how the feature interaction had been formed. This information helped us decide which part in the source code should be corrected.

We are now on the way to establishing a tool support to interactively visualize deviant effects as feature interactions. We will evaluate the feasibility and usefulness of our approach by applying it to other deviant effect examples.

ACKNOWLEDGMENTS

We are deeply grateful for useful discussions with Naoya Nitta, Masahiro Nakajima, and Professor Norihiro Hagita. This work was partially supported by MEXT/JSPS KAKENHI [Grant-in-Aid for Challenging Exploratory Research (No.23650016), Scientific Research (C) (No.24500079), Scientific Research (B) (No.23300009)], and Kansai Research Foundation for technology promotion.

REFERENCES

- [1] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, June/July 1988.
- [2] S. Sparks, K. Benner, and C. Faris, "Managing object-oriented framework reuse," *IEEE Computer*, vol. 29, no. 9, pp. 52–61, 1996.
- [3] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *ECOOP*, 2010, pp. 2–25.
- [4] M. Nakamura, H. Igaki, and K. ichi Matumoto, "Feature interactions in integrated services of networked home appliances," in *Feature Interactions in Telecommunications and Software Systems*, 2005, pp. 236–251.
- [5] M. Wilson, M. Kolberg, and E. H. Magill, "Considering side effects in service interactions in home automation - an online approach," in *ICFI*, 2007, pp. 172–187.
- [6] A. Lienhard, *Dynamic Object Flow Analysis*. Lulu.com, 2008.
- [7] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing java programs," in *International Conference on Software Engineering*. IEEE, 2004, pp. 512–521.
- [8] I. Kume and E. Shibayama, "Feature interactions in object-oriented effect systems from a viewpoint of program comprehension," in *International Conference on Feature Interactions*, 2009.