A Case Study of Dynamic Analysis to Locate Unexpected Side Effects Inside of Frameworks

Izuru Kume, Nara Institute of Science and Technology, Ikoma, Japan

Masahide Nakamura, Kobe University, Kobe, Japan

Naoya Nitta, Konan University, Kobe, Japan

Etsuya Shibayama, The University of Tokyo, Tokyo, Japan

ABSTRACT

Recently many frameworks are deployed without proper documents to explain their correct usage. In the absence of proper documents, application developers often write code to call a framework API in a wrong way. Such a wrong API call tends to bring about a failure after its complex chain of infection inside of a framework. The complexity and the lack of implementation knowledge about a framework make it difficult for application developers to debug this kind of failure. In the preceding study the authors focused on unexpected side effects that are caused by wrong API calls and bring about failures, and developed a dynamic analysis technique to detect such side effects. In this paper, the authors introduce a case study to find a wrong API call using our technique.

Keywords: *Application Framework, Debug, Dynamic Analysis, Object-Oriented Programming Language, Program Understanding, Side Effect, Software Engineering*

INTRODUCTION

Recently many frameworks are used in software development without proper documents to explain their correct usage (Shull, Lanubile & Basili, 2000). As a result, application developers often write code to call APIs provided by frameworks in wrong ways (Monperrus & Mezini, 2013). Several static analysis techniques (Monperrus & Mezini, 2013; Mishne, Shoham & Yahav, 2012) are proposed to solve this problem, but they don't cover such wrong API calls that maintainers can find them faulty only by examining their runtime conditions such as the reference structure among involved objects, the timing of inversion of controls to trigger the API calls, and etc.

DOI: 10.4018/IJSI.2015070103

We pursue a method to debug wrong API calls as defects. In general, debugging a failure requires an examination of source code and its execution. Such a task is necessary in finding a chain of infection (Zeller 2009). In this task, maintainers are required to find erroneous states based on their implementation knowledge of the system under debugging. For example, they should find that some value of a local variable is faulty, or a method is invoked at a wrong timing.

Maintainers of a framework application, who are application developers and suspect a wrong API call, try to trace back the execution from a failure to this wrong API call. Because such a wrong API call produces an erroneous state inside of the framework, they have to examine the source code of the framework and its runtime states. Usually they are new to the implementation details of the framework, and thus their task to detect a chain of infection inside of the framework is very difficult and time consuming.

To cope with this problem, we leverage possibly unexpected side effects which seem to be hidden from frameworks, and cause failures via outdated objects' state. In our preceding study (Kume, Nitta, Nakamura & Shibayama, 2014), we developed a dynamic analysis technique to detect such hidden updates and uses of outdated states in a program execution trace. In this paper, we introduce a case study where we found wrong API calls by detecting an unexpected side effect using our technique.

The rest of this paper is as follows: In section PRELIMINARY, we introduce basic concepts of framework applications, and we also explain the difficulty to debug wrong framework API calls. In section PROPOSED APPROACH, we explain an overview of our dynamic analysis technique. In section CASE STUDY, we introduce our case study, and discuss the usefulness and limitation of our technique in section DISCUSSION. Section RELATED WORK is for explaining our related work, and we state our conclusion in CONCLUSION.

PRELIMINARY

Operations, Statements, and Dependency

We assume that frameworks and their applications are implemented in Java language. Java objects consist of class instances and arrays. *Operations* on objects are method invocations (except for static methods), and accesses to their instance variables or array components. An operation is expressed as a statement or an expression in a statement.

Parameters of a method invocation consist of its receiver (if any) and arguments. We call instance variables and array components *persistent variables*. When a persistent variable of an object is accessed to assign or get a value, then we say that the object is used as a *carrier* of the value, and that the object carries the value.

In addition to ordinary dependency among statements (Tip, 1995), we introduce new kinds of dependency among operations and statements. A get operation on a persistent variable depends on the operation that assigned the got value to the persistent variable. A method invocation whose receiver is not null executes the method body bound at runtime based on the receiver class. Thus method receivers work in a similar way to operands of conditional branching statements.

A value carrier itself may have been carried by another object, which is the carrier of the carrier of the value. We may further get the carrier of the carrier of the carrier of the value. For a carried value, we can thus obtain a sequence of references of persistent variables which have brought the value. We call such a sequence *a reference path to the value or its carrier.*

Application Frameworks

We categorize classes and methods in a framework application. We call classes (methods) contained in a framework framework classes (framework methods). We call those classes (methods) other than framework classes application-specific classes (application-specific

methods) (Kume, Nakamura & Shibayama, 2012). Application-specific classes implement application-specific features. We also use terms hot spots, template methods, and hook methods, which are introduced in (Pree, 1994) in order to discuss framework architectures. We call a framework method that represents a hot spot a hot spot method, and its class a hot spot class.

Application developers implement application-specific classes that inherit hot spot classes and override their hot spot methods. Application developers should understand (1) which hot spot classes to inherit, (2) at which timing overriding methods are invoked, and (3) when and how to invoke framework methods which implement the framework APIs (Heydarnoori, Czarnecki, Binder & Bartolomei, 2012).

Examples of framework API calls are found in dependency injection (Fowler, 2004) such as GUI initialization. We name hot spot classes and those framework classes that implement framework APIs *exposed framework classes*. Hot spot methods and API methods are named *exposed framework methods*. Exposed framework classes model basic entities in the application domain of a framework, and their methods represent the features of the entities.

An exposed class is an application-specific class, an exposed framework class, or a Java library class. An exposed method is a method of an exposed class and is visible in application-specific classes. Application developers usually understand the entities represented by exposed framework classes, and the features implemented by their methods.

API methods are necessary for an application-specific part to setup a correct state inside of a framework which results in a correct effect in later execution. The correspondence between setup calls and their results is hidden inside of the framework. Understanding such correspondence requires the implementation knowledge of the framework. Therefore, it is difficult for application developers, who usually lack such knowledge, to find a wrong API call by examining the execution inside of the framework.

PROPOSED APPROACH

Assumptions

Our approach aims at supporting application developers who try to debug a wrong API call in a framework application. We assume that application developers don't have proper documents that explain correct ways to call framework APIs. We assume such a case that a wrong API call results in an unexpected side effect, and this side effect causes a failure. In other words, the chain of infection triggered by the wrong API call involves the unexpected side effect.

The application developers understand the application domain entities and their representation by exposed framework classes. They understand relationships among exposed framework classes and their instances in terms of the domain concepts. They can explain the usages of parameters of exposed methods in terms of the domain concepts.

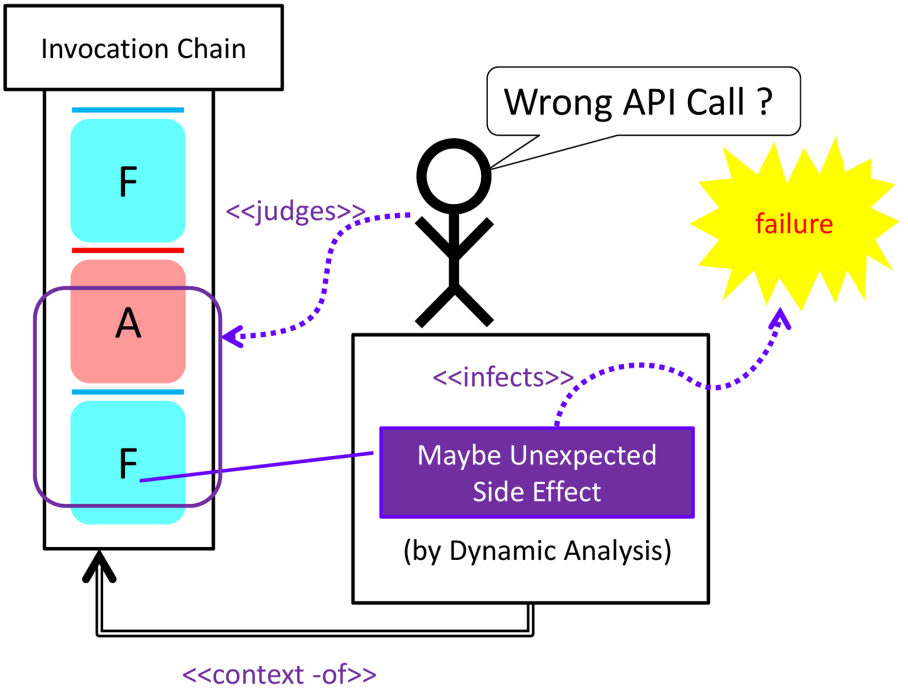
For example, they can explain the internal process accomplished by an exposed method like this: "This framework method removes a graph edge, which is the method receiver, from the current diagram." They also understand the class relationships in terms of the domain concepts. Thus they can explain the meaning of reference paths like this: "A graph edge connects this node with another."

Basic Idea

The basic idea of our approach is to find wrong API calls by detecting their unexpected side effects which cause failures. (Figure 1) Our approach enables application developers to avoid a time consuming task to track the chain of infection from a failure back to a wrong API call. In general, framework applications have many side effects at runtime. Therefore, we must support application developers to find unexpected side effects that lead them to wrong API calls.

For this support, we use a dynamic analysis tool developed in our preceding study (Kume,

Figure 1. Method overview



Nitta, Nakamura & Shibayama, 2014). Given a trace of a failed execution, our tool detects side effects which cause a failure and match a behavioral pattern to suggest their unexpectedness. Our tool also detects behavioral 'bad smells'. Such bad smells are related to detected side effects, and suggest that some of them are really unexpected. Detected side effects and bad smells are called *symptoms* in our preceding study (Kume, Nitta, Nakamura & Shibayama, 2014).

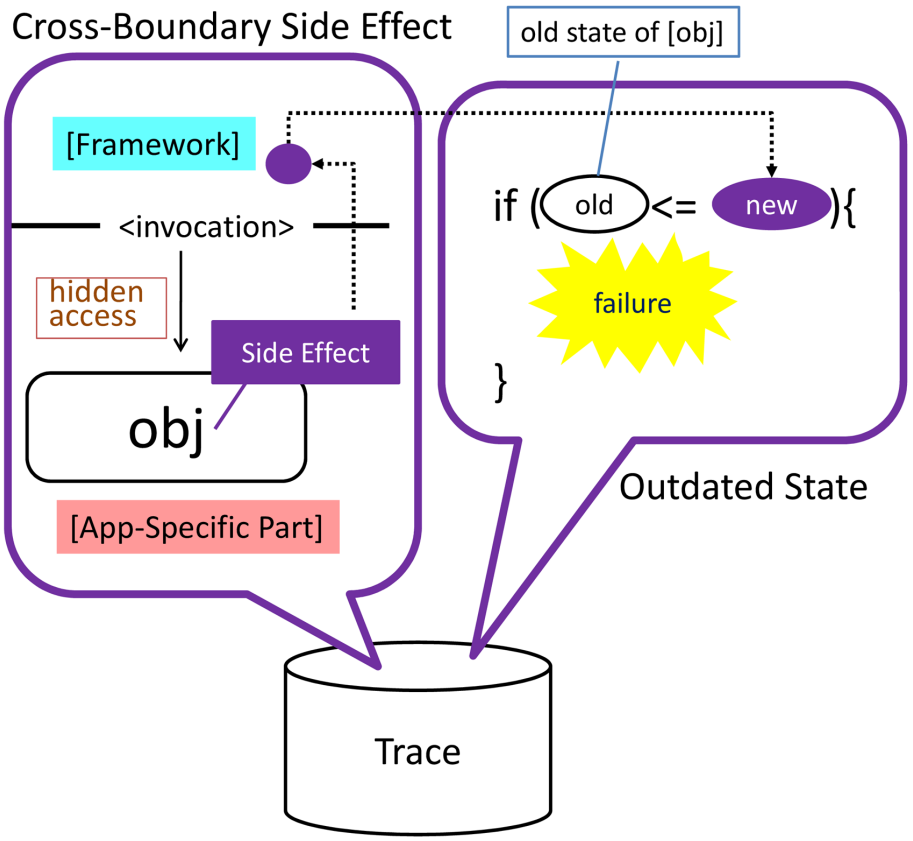
Symptoms

A symptom is a collection of executed statements and operations on objects that satisfy a particular condition. We defined three conditions for matching candidates of unexpected side effects and two kinds of bad smells. The candidates are called *cross-boundary side effects*, and the bad smells *outdated states* and *aliasings*.

A cross-boundary side effect consists of an assignment operation on a value carrier, a method invocation, a reference path to the carrier, and statements to form a data flow. We illustrate an example of cross-boundary side effect in the left side of Figure 2. Here, the candidate side effect is brought by an assignment operation on a carrier obj. The assignment operation is executed under an application-specific method, which has been invoked in a framework method.

Under the application-specific method, the carrier is accessed via a reference path from one of the method parameters. After this method returns to the framework method, the assigned value or another value which depends on the assigned value is used there.

Figure 2. Symptoms that suggest an unexpected side effect



From a lexical viewpoint, the access to obj and therefore the assignment operations on one of its persistent variables are 'hidden' from the framework method. If the framework developers didn't assume such a side effect, then it is an unexpected one made by this application-specific method[1].

An outdated state symptom represents a kind of interference by a side effect at conditional branching. It is defined as the dependency of a conditional branching statement both on an old and a new value of the same persistent variable. An outdated state combined with a cross-boundary side effect should be suspicious because the side effect makes a wrong branching based on the unexpectedly updated state. (Figure 2)

An aliasing symptom represents the existence of more than one reference paths to the same value carrier. The reference path in an aliasing may tell application developers some anomaly using application developers' domain knowledge.

Judgment of API Calls

For a detected cross-boundary side effect, application developers can examine the invocation chain under which the side effect is executed. They should judge if there is an API call which introduces a mismatch between the intention of

a template method and actual behavior of its hook methods.

A template method intends that its hook methods take some roles in its whole task. The actual behavior of each of the hook methods must match its intended role. Application developers need to check if their actual behaviors match their intended roles. This judgment requires a technique to understand a framework's internal behavior without its implementation knowledge.

We believe that reference paths among instances of exposed classes give application developers enough information for their judgment. Exposed classes of value carriers in a reference path combined with their mutual reference structure will help application developers understand as which a framework regards the obtained object. Such information about value carriers and hook method parameters enables application developers to understand the intended roles of side effects and invoked hook methods. We have shown such a case study in our preceding study (Kume, Nakamura, & Shibayama, 2012).

Implementation

Our analysis tool generates an execution trace of a framework application under debugging with its byte code under instrumentation using BCEL[2]. A generated trace contains executed statements and operations as well as dependency among them similar to that of Wang & Roychoudhury (2004). In addition, the trace categorizes classes and methods as we explained in section PRELIMINARY.

Obtaining execution traces of Java Collection Framework classes may be possible. However, our tool does not do it for the original JDK classes. Instead, our tool substitutes the original JDK library classes with corresponding Open JDK classes before instrumentation. It is for avoiding a possible violation of the license terms.

Our tool starts an analysis process for an uncaught exception. It detects symptoms on which the exception depends. It depicts how the exception statement depends on each of the detected symptoms. For this purpose, our tool abstracts the invocation chain under which the exception is thrown. The abstraction is based on an exposed method in the invocation chain. The exposed method is the 'nearest' one in the sense that no other exposed methods are executed under it.

The invocation chain is abstracted in terms of (1) the control flow which invokes the exposed method, (2) the parameters of the method, and (3) the control flow which executes the exception throwing statement.

A symptom is related to a particular value carrier which plays an important role in its behavior. As for a cross-boundary side effect, it is the object which is accessed from a method parameter and has its persistent variable assigned a value. For an outdated state symptom, it is the carrier of an old and a new value. For an aliasing symptom, it is the value carrier accessed via the multiple reference paths. Our tool summarizes classes of such value carriers of depicted symptoms. In addition, our tool outputs reference paths to these carriers in detected symptoms.

Our tool assigns a unique ID number to each of executed operations and referenced objects. Application developers can use these ID numbers in order to make correspondence between user inputs and invoked event handlers. Thus they can relate visualization results to runtime objects[3] by examining these output results.

CASE STUDY

Example Application

We have debugged a wrong API call in a simple UML editor[4] built on GEF[5], a practical open source framework for graph editors. The framework and this UML editor were developed by a third party. The GEF version used in this experiment had been outdated already, and a newer and bug fixed version has been released. GEF itself is built on Swing, and Swing events are passed to an application specific part through GEF's event handlers.

Figure 3. Example application

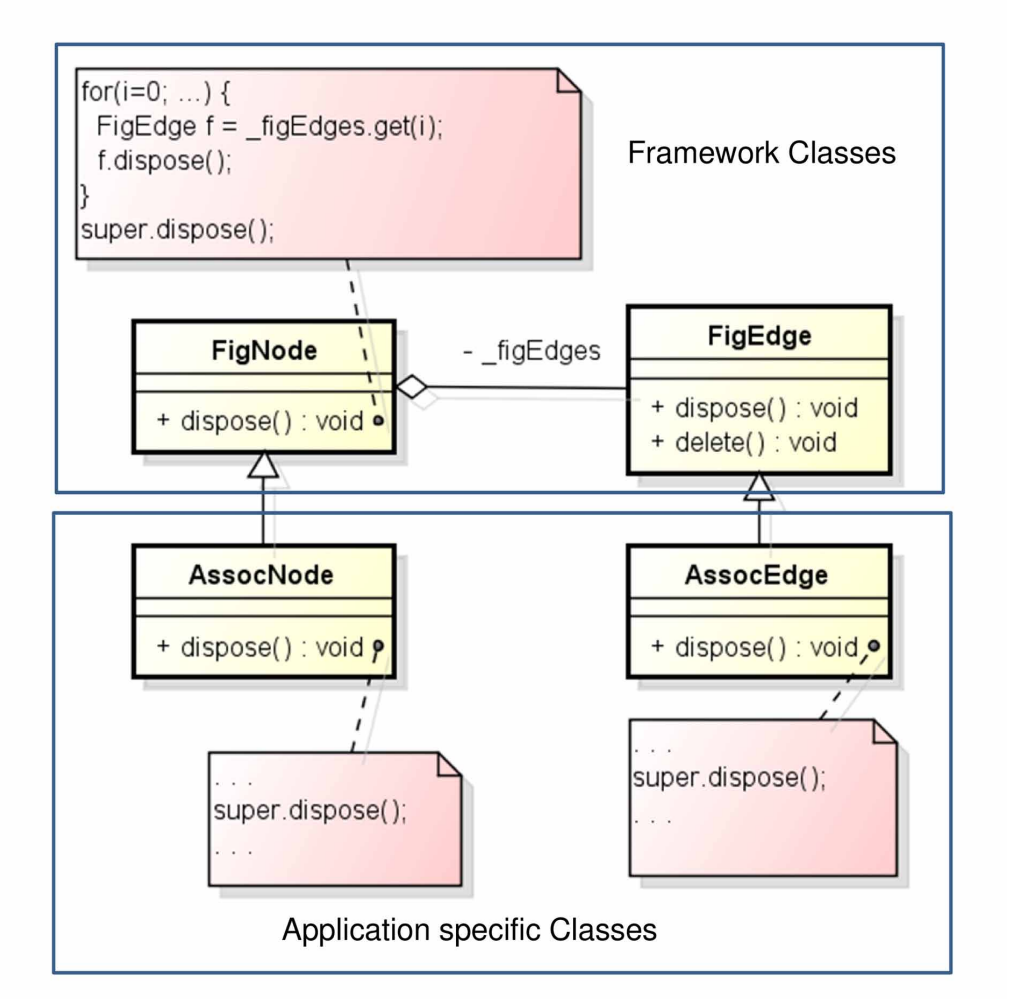
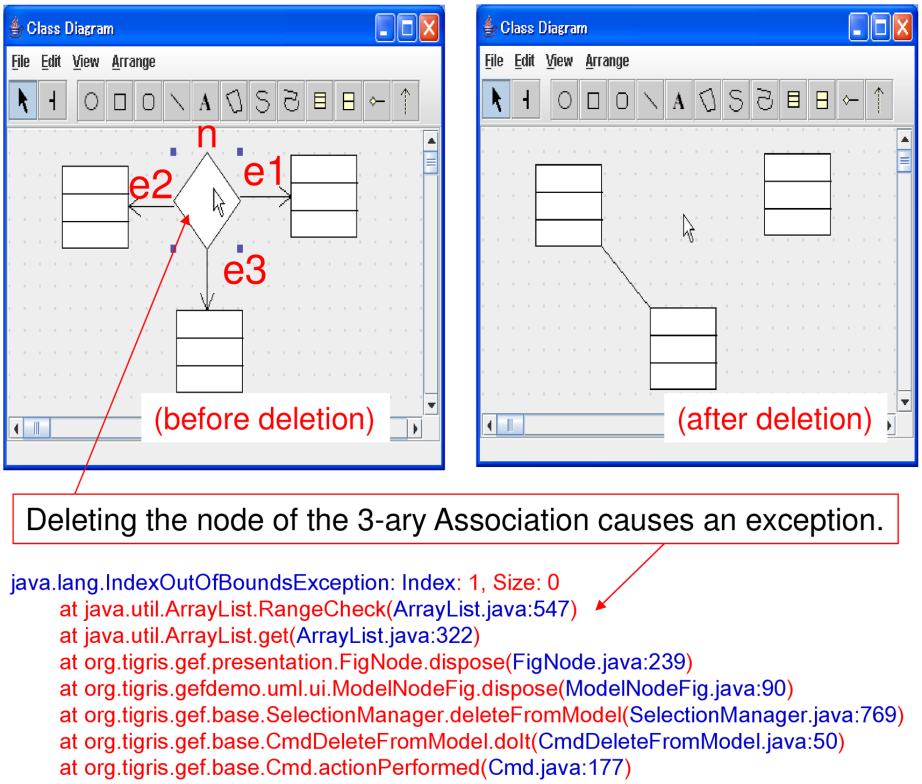


Figure 3 shows the architecture of this application. Some of the class names and a part of class hierarchy are abbreviated in order to simplify our explanation. Two framework classes FigNode and FigEdge represent general graph nodes and their edges, respectively. They are GEF's hot spot classes. They implement a hot spot method dispose() which is invoked in a template method in response to a user's operation to delete a graph node or a graph edge.

Application specific classes AssocNode and AssocEdge represent nodes and edges of UML n-ary associations, respectively. They inherit FigNode and FigEdge respectively, and override dispose(). The overriding methods are actually invoked inside of GEF at runtime. These methods are used to give chances for instances of application specific classes to perform their own deletion procedures. The pseudo code in figure 3 contains invocations of overridden dispose() (expressed by super.dispose()). These are API calls to request the framework to graphically remove UML model elements from a diagram.

Figure 4 shows a failure example in editing a UML diagram. This failure can be reproducible by the following user operations. First,

Figure 4. Failure example



create three UML Classes and connect them with a ternary UML Association. Then select the node of the association, and press DEL key. At the key press, an error message is shown, and the association is not deleted completely as is shown in Figure 4. From the error message, we can see that an exception is thrown because this program tries to obtain an element from an empty list, which is an ArrayList instance.

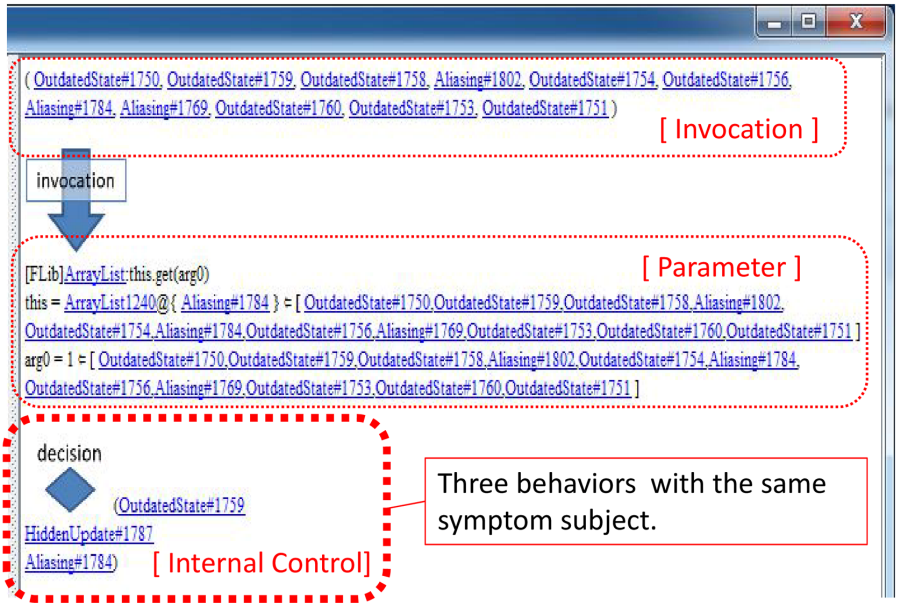
Detected Symptoms

Figure 5 shows the visualization result generated by our tool[6] that depicts the dependency among detected symptoms, the invocation ArrayList.get(int index), and the exception statement. Each of detected symptoms is denoted by a tag name and a unique ID number. Figure 6 summarizes classes of carriers of these symptoms.

HiddenUpdate tag represents cross-boundary side effects. OutdatedState and Aliasing tags represent outdated state symptoms and aliasing symptoms, respectively. Figure 5 shows that the tool can detect only one cross-boundary side effect (HiddenUpdate#1787).

The control flow inside of method get(int index) depends on this side effect. We saw that its carrier is an instance of ArrayList from the table in Figure 6. By examining the reference paths involved in symptoms, we also saw that this ArrayList instance is also the carrier of OutdatedState#1759 and Aliasing#1784.

Figure 5. Detected symptoms



Judgment of API Calls

The detected cross-boundary side effect and the exception throwing statement are executed under the same event handler. The event handler is invoked by Swing in response to the press of DEL key. Figure 7 depicts the least invocation tree to contain the side effect and the exception throwing statement. The assignment operation is denoted with ID [8], while the exception statement with ID [0]. Capital letters before ID numbers are to designate method categories explained in section PRELIMINARY. A, F, and J represent an application-specific method, a framework method, and a Java library method, respectively.

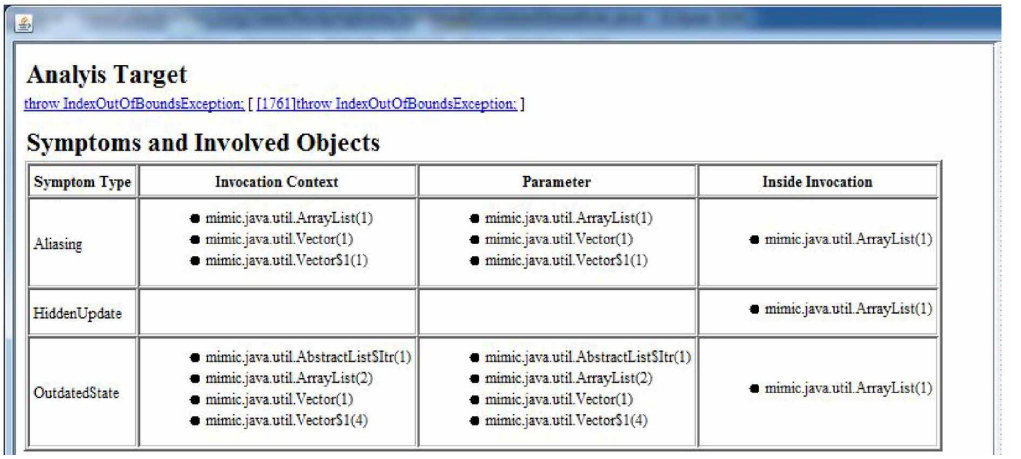
Note that A[4] and A[9] make API calls super.dispose(). A[4] overrides the hot spot method FigNode.dispose(), and A[9] overrides FigEdge.dispose(). We checked if their behaviors match the intentions of their invokers, which are F[3] and F[5] respectively. By making correspondence between our operations on UML model elements in Figure 4 and invoked event handlers which create them, we could find that the model elements are used as parameters of the methods in Figure 7. In Figure 7, we use the names of the method parameters (n, e1, e3) that are shown in Figure 4.

By examining the reference paths among the method parameters, we could recognize the following process. First, framework method F[5] obtains an edge e1 from the given node n, and tries to dispose the edge. In A[9] the edge e1 obtains its sibling edge e3 from node n and remove it by an API call F[10]. The detected cross-boundary side effect represents the effect of this removal operation on the ArrayList instance.

It is obvious that this side effect is unexpected for F[5] because it tries to invoke get(int index) on this object after method A[9] returns. Therefore, we find a mismatch between the intention of F[5] and the behavior of A[9].

Correction of this mismatch was a bit complicated. Method A[9] has a rationale to

Figure 6. Carrier classes of detected symptoms



Analyis Target

throw IndexOutOfBoundsException; [[1761]throw IndexOutOfBoundsException;]

Symptoms and Involved Objects

Symptom Type	Invocation Context	Parameter	Inside Invocation
Aliasing	● mimic.java.util.ArrayList(1) ● mimic.java.util.Vector(1) ● mimic.java.util.Vector$1(1)	● mimic.java.util.ArrayList(1) ● mimic.java.util.Vector(1) ● mimic.java.util.Vector$1(1)	● mimic.java.util.ArrayList(1)
HiddenUpdate			● mimic.java.util.ArrayList(1)
OutdatedState	● mimic.java.util.AbstractList$Itr(1) ● mimic.java.util.ArrayList(2) ● mimic.java.util.Vector(1) ● mimic.java.util.Vector$1(4)	● mimic.java.util.AbstractList$Itr(1) ● mimic.java.util.ArrayList(2) ● mimic.java.util.Vector(1) ● mimic.java.util.Vector$1(4)	● mimic.java.util.ArrayList(1)

remove sibling edges. Such a removal is necessary when a ternary association becomes a binary association by removing an edge. Such a change is accompanied by the change of its shape. A[9] works well for such user operations. Finally we revised A[4] so that it removes all edges of the given node before invoking F[5].

DISCUSSION

As we found in the previous section, the call of API method F[10] in A[9] brings about an unexpected side effect if the following two conditions are satisfied: (1) A[9] is invoked in F[5], and (2) the FigEdge object is an instance of AssocEdge. Therefore, this mismatch required us to examine the inversion of control (invocation of A[9] by F[5]) and the dynamic binding by AssocEdge. Static analysis methods don't suit this kind of examination in general.

It is also difficult to find the defect, the wrong API call in A[4], by using an existing debugger because our lack of the implementation knowledge of GEF. Our tool saved our effort to find the unexpected side effect from the failure. We only had to examine the invocation chain under which the list object became empty, and understood the meaning of the invocation chain in terms of the involved objects, which are the node and its edges.

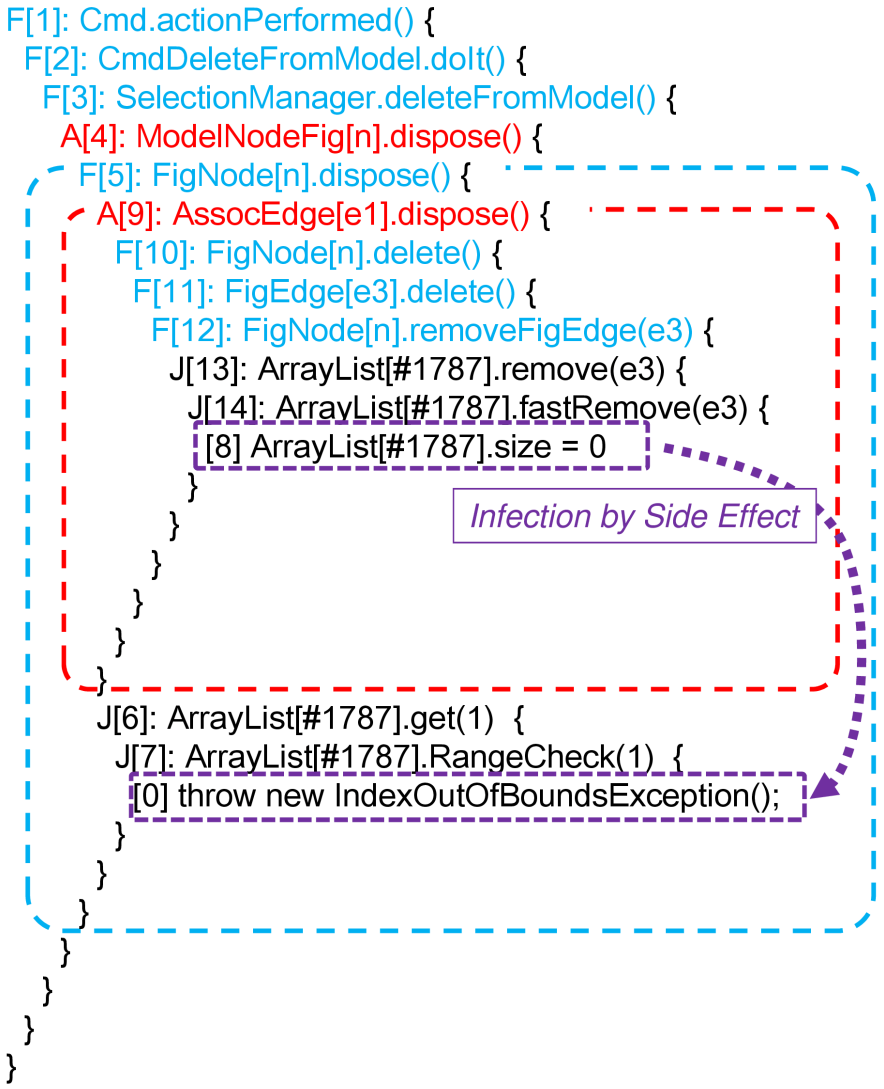
The application under debugging has more than 100,000 lines of code, and its invocation structure is too complex to manually examine step by step using an existing debugger. The invocation tree in Figure 7 might seem to be a little complex, although many invoked methods under the root event handler are omitted. Many of these omitted invocations should have been included there if we had adopted some classical slicing technique in Tip (1995).

From our experimental result, we can point out several factors that made our approach applicable in practice. First, the application runs in a event driven style, therefore we could easily correspond object creations to users' operations by examining the event handlers. Second, the classes of the parameter objects in Figure 7 represent basic entities of the graph structure except for the ArrayList instance, which made us easily understand their roles in the framework.

As for another factor, we point out the clarity of the specification of the exposed framework methods in Figure 7. We also point out that the reference paths which we examined contain only instances of exposed classes[7]. Therefore, we could easily understand the structural relationship among the method parameters.

As for the limitation of our approach, we must admit that it is not applicable to so called missing method calls problem (Monperrus,

Figure 7. Invocation context



Bruch & Mezini, 2010), a problem that application-specific methods often forget necessary API calls to their frameworks.

Our analysis tool should implement a sophisticated user interface to provide an abstract view of reference paths. In our case, our tool summarized the classes of the symptom subjects, all of which belong to Java Collection Framework as we can see in Figure 7. We could not understand their roles in the application until we related them with instances of exposed framework classes.

We could not avoid the examination of the framework code, although we spent few efforts for this task. We had to read the code of dispose() of FigNode in order to ensure that it throws the exception when it tries to obtain the second edge of the nod. We couldn't understand

the meaning of such a 'failed attempt' without examining the source code.

RELATED WORK

A wrong API call results in a kind of deviant behavior (Engler, Chen, Hallem, Chou & Chelf, 2001). A deviant behavior is brought by an implementation without the knowledge about "correctness rules the system must obey" (Engler, Chen, Hallem, Chou & Chelf, 2001). Such a wrong call in a framework application is called deviant code (Monperrus, Bruch & Mezini, 2010). Existing approaches of deviant code detection (Monperrus & Mezini, 2013; Mishne, Shoham & Yahav, 2012; Wasylkowski, Zeller & Lindig, 2007) are based on static analysis of method invocations or operations on objects.

Monperrus and Mezini (Monperrus & Mezini, 2013) propose a static analysis of co-occurrence of invoked methods. Wasylkowski and et al. (Wasylkowski, Zeller & Lindig, 2007) propose a method to extract object usage patterns. These approaches suit such cases that any correct code obeys an implicit pattern of co-occurrence of method invocations, or a set of implicit temporal properties on an object usage.

Such mining approaches will be successful if their analysis targets are method invocations in a single method (Monperrus & Mezini, 2013), or based on the usages of "the same abstract object" in a method (Wasylkowski, Zeller& Lindig, 2007). These limitations are necessary to draw a line between correct behaviors and deviant behaviors. If we need a mining method for different purposes, such as finding a pattern of API calls to implement a feature, we can adopt less limited approaches such as temporal specification mining by Mishne and et al. (Mishne, Shoham & Yahav, 2012).

As for the example in CASE STUDY section, we had to examine the structure of the method invocation tree as well as the complex reference structure among the node and its edges that are operated in the methods invoked there. From the viewpoint of framework usage, it depends on the timing of inversion of controls whether the removal of sibling edges becomes a deviant behavior or not. As a result, resolving this example required program comprehension with respect to a complex structure of method invocations and object references. Such a kind of comprehension is too complex and semantic dependent for existing mining methods.

We can regard a deviant behavior as a chain of infection caused by its deviant code, and pursue a supporting method to debug it. In general, finding a chain of infection is a time consuming task, and existing debuggers and debugging techniques (Lencevicius, 2000; Lencevicius, Hölzle & Singh, 2003; Zhang, Gupta & Gupta, 2006) assume maintainers' knowledge about the program code under debugging. However, a deviant behavior contains an infection process inside of the framework of the application under debugging. Because application developers are usually new to the implementation of their frameworks, existing approaches that depends on maintainers' implementation knowledge don't seem very promising.

Framework applications normally use side effects inside of their frameworks in order for application specific features to work correctly. However, when side effects are involved in a chain of infection, it is a time consuming task to examine these side effects by existing debugging tools. Existing debugging tools cannot trace back directly to the methods which have already been popped from the stack, although causes of many errors are found there (Lienhard, Gîrba & Nierstrasz, 2008).

Omniscient debugger (Pothier, Tanter & Piquer, 2007) and backward-in-time debugger (Lienhard, Gîrba & Nierstrasz, 2008) are studied in order to cope with this problem. They are very promising if maintainers understand implementation details of the system under debugging. However, as for debugging deviant behaviors in framework applications, we also face the problem that maintainers don't understand implementation details of their frameworks.

Our previous work (Kume, Nakamura & Shibayama, 2012; Kume, Nitta, Nakamura & Shibayama, 2014) aims at an abstraction method that supports maintainers to understand

the behaviors inside of a framework without examining its implementation details. Deviant behaviors in framework applications can be thought as feature interactions between frameworks and their application-specific parts. Thus we expect that we can apply existing method in this area (Nakamura, Igaki, & Matumoto, 2005; Leung, 2007) to resolve our problem.

Behavioral abstraction and visualization are keys to success in finding and understanding deviant behaviors without examining implementation details. Dynamic Object Flow Analysis (Lienhard, 2008; Lienhar, Greevy & Nierstrasz, 2007; Lienhard, Gîrba, & Nierstrasz, 2008) makes it possible to analyze aliasing, which are references to an object at runtime. This method well suits dependency analysis for regression testing, for example. However, its visualization seems not so successful showing the essential difficulty to deal with the complexity of aliasing history to even a single object.

Dynamic Object Process Graph (Quante, 2008; Quante & Koschke, 2008) aims at analyzing a particular object from the viewpoint of its operations. Although this method focuses on a single object, it well captures the abstracted dynamics of its state changes. Such an abstraction style will be useful when we extend our method so that it can deal with dynamics among multiple objects.

Whyline (Ko & Myer, 2009) is another successful example of visualization for debug. Whyline is implemented as a sophisticated debugging tool which enables maintainers to easily examine why a statement was executed (or why a statement was not executed). The tool has a well-designed GUI by which maintainers relate their questions about infection directly to visible outputs of the system under debugging. The cognitive study under the GUI design is very suggestive when we revise the GUI design of our prototype tool.

Currently symptoms are hard-coded in our prototype tool. In future, we will re-design our tool so that maintainers can define their own symptoms. For this purpose, we need to introduce a pattern description language to extend symptom analysis. Languages for query or pattern matching on traces (Goldsmith, O'Callahan & Aiken, 2005; Martin, Livshits & Lam, 2005) are promising for specifying behavioral patterns for symptom detection.

CONCLUSION

In this paper, we proposed a dynamic analysis technique that supports debugging of a failure, which was brought about by an unexpected side effect caused by a wrong framework API call. Our technique is applicable for such a case that existing static analysis methods are not applicable. Our technique aims at freeing application developers from time consuming tasks to track back unexpected side effects along a chain of infection inside of a framework. We showed an experimental result to apply our technique to a task of debugging a deviant behavior in a practical framework application, and discussed its usefulness and limitation.

ACKNOWLEDGMENT

We are deeply grateful for useful discussions with 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), and Scientific Research (B) (No.23300009)].

REFERENCES

Engler, D., Chen, D. Y., Hallem, S., Chou, A., & Chelf, B. (2001). Bugs as deviant behavior: a general approach to inferring errors in systems code. In the eighteenth ACM symposium on Operating systems principles (SOSP 2001) (pp. 57-72). Chateau Lake Louise, Banff, Canada: ACM Press. doi:10.1145/502039.502041

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. *Martin Fowler*, Retrieved October 10, 2011, from http://www.martinfowler.com/articles/injection.html

Goldsmith, S., O'Callahan, R., & Aiken, A. (2005). Relational Queries over Program Traces. In *ACM OOPSLA 2005* (pp. 385–402). San Diego, California, USA: ACM Press.

Heydarnoori, A., Czarnecki, K., Binder, W., & Bartolomei, T. T. (2012). Two studies of framework-usage templates extracted from dynamic traces. *IEEE Transactions on Software Engineering*, *38*(6), 1464–1487. doi:10.1109/TSE.2011.77

Ko, A., & Myer, B. (2009). Finding Causes of Program Output with the Java Whyline. In *The 27th ACM Conference of Human Factors in Computing Systems (SIGCHI 2009)* (pp. 1569–1578). Atlanta, Georgia, USA: ACM Press. doi:10.1145/1518701.1518942

Kume, I., Nakamura, M., & Shibayama, E. (2012). Toward comprehension of side effects in framework applications as feature interactions. In *The 19th IEEE Asia-Pacific Software Engineering Conference (APSEC 2012)* (pp. 713-716). Hong Kong: IEEE. doi:10.1109/APSEC.2012.128

Kume, I., Nitta, N., Nakamura, M., & Shibayama, E. (2014). A dynamic analysis technique to extract symptoms that suggest side effects in framework applications. In *The 29th ACM Symposium On Applied Computing* (pp. 1176-1178). Gyeongju: ACM Press. doi:10.1145/2554850.2555123

Lencevicius, R. (2000). *Advanced Debugging Methods*. Dordrecht, the Netherlands: Kluwer Academic Publishers. doi:10.1007/978-1-4419-8774-7

Lencevicius, R., Hölzle, U., & Singh, A. K. (2003). Dynamic query-based debugging of object-oriented programs. *Automated Software Engineering*, *10*(1), 39–74. doi:10.1023/A:1021816917888

Leung, W. F. (2007). Program entanglement, feature interaction and the feature language extensions. Computer Networks. *The International Journal of Computer and Telecommunications Networking*, *51*(2), 480–495.

Lienhard, A. (2008). *Dynamic Object Flow Analysis*. Lulu.com.

Lienhard, A., Gîrba, T., & Nierstrasz, O. (2008). Practical Object-Oriented Back-in-Time Debugger. In *J. Vitek (Ed), The 22nd European Conference on Object-Oriented Programming (ECOOP 2008): LNCS 5142* (pp. 592-615). Paphos, Cyprus: Springer.

Lienhard, A., Greevy, O., & Nierstrasz, O. (2007). Tracking Objects to Detect Feature Dependencies. In 2007 IEEE 15th International Conference on Program Comprehension (ICPC) (pp. 59-68). Banff, AB, Canada: IEEE. doi:10.1109/ICPC.2007.38

Martin, M., Livshits, B., & Lam, M. S. (2005). Finding Application Errors and Security Flaws using PQL: A Program Query Language. In *ACM OOPSLA 2005* (pp. 365–383). San Diego, California, USA: ACM Press. doi:10.1145/1094811.1094840

Mishne, A., Shoham, S., & Yahav, E. (2012). Typestate-based semantic code search over partial programs. In *ACM OOPSLA 2012* (pp. 997–1916). Tucson, Arizona, USA: ACM Press. doi:10.1145/2384616.2384689

Monperrus, M., Bruch, M., & Mezini, M. (2010). Detecting missing method calls in object-oriented software. In T. D'Hondt (Ed), *The 24th European Conference on Object-Oriented Programming (ECOOP 2010): LNCS 6183* (pp. 2–25). Maribor, Slovenia: Springer. doi:10.1007/978-3-642-14107-2_2

Monperrus, M., & Mezini, M. (2013). Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology, 22(1)*. 7:1-7:25.

Nakamura, M., Igaki, H., & Matumoto, K. (2005). Feature Interactions in Inegrated Services of Networked Home Appliances. In S. Reiff-Marganiec & M. Ryan (Eds.), *Feature Interactions in Telecommunications and Software Systems VIII (ICFI'05)* (pp. 236–251). Leicester, UK: IOS Press.

Pothier, G., Tanter, É., & Piquer, J. (2007). Scalable Omniscient Debugging. In *ACM OOPSLA 2007* (pp. 535–552). Nashville, Tennessee, USA: ACM Press.

Pree, W. (1994). *Design Patterns for Object-Oriented Software Development*. Boston, Massachusetts, USA: Addison-Wesley.

Quante, J. (2008). Do Dynamic Object Process Graphs Support Program Understanding? – A Controlled Experiment. In 2008 IEEE 16th International Conference on Program Comprehension (ICPC 2008) (pp. 73-82). Amsterdam, The Netherlands: IEEE.

Quante, J., & Koschke, R. (2008). Dynamic object process graphs. *Journal of Systems and Software*, *81*(4), 481–501. doi:10.1016/j.jss.2007.06.005

Shull, F., Lanubile, F., & Basili, V. R. (2000). Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering*, *26*(11), 1101–1118. doi:10.1109/32.881720

Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, *3*, 121–189.

Wang, T., & Roychoudhury, A. (2004). Using compressed bytecode traces for slicing java programs. In *The 26th IEEE International Conference on Software Engineering* (pp. 512–521). Scotland, UK: IEEE.

Wasylkowski, A., Zeller, A., & Lindig, C. (2007). Detecting Object Usage Anomalies. In *The 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE 2007)* (pp. 35-44). Dubrovnik, Croatia: ACM Press. doi:10.1145/1287624.1287632

Zeller, A. (2009). Why Programs Fail, Second Edition: A Guide to Systematic Debugging. Burlington, Massachusetts, USA: Morgan Kaufmann.

Zhang, X., Gupta, N., & Gupta, R. (2006). Pruning dynamic slices with confidence. In *The 27th ACM Conference on Programming language design and implementation (PLDI 2006)*. (pp. 169–180). Ottawa, Canada: ACM Press. doi:10.1145/1133255.1134002

ENDNOTES

1. Remember that application developers are often different from framework developers.
2. http://commons.apache.org/proper/commons-bcel/
3. Java Whyline (Ko & Myer, 2009) implements this feature with its well-designed User Interface.
4. http://gefdemo.tigris.org/
5. http://gef.tigris.org/
6. We add annotations surrounded by red lines by hand for an explanation.
7. An array class which component type is Object is included in these classes. We can easily guess it is used to implement ArrayList.