2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing

A Feature Model of Framework Applications

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

Naoya Nitta
Faculty of Intelligence and Informatics, Konan University
Higashinada, Kobe, Japan
Email: n-nitta@konan-u.ac.jp

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

***Abstract*—Learning how to use application frameworks effectively becomes important in their widespread use in software development. Learning frameworks is often difficult because of lack of their documentation and their complexity. In order to help framework learning, we propose a behavioral model, called *feature component model*, which abstracts internal behaviors of framework applications in terms of their behavioral characteristics such as inversion of controls. We apply our behavioral model to an example misuse of a practical framework developed by a third party in order to show its practical usefulness.**

***Keywords*—application frameworks; framework learning; feature model; feature interactions; program comprehension;**

I. Introduction

Learning how to use application frameworks effectively becomes important because of their widespread use in software development. Many framework products are deployed without proper documentation about their correct use. Therefore their users, application developers, must often learn their correct use from the program code of example applications [1].

In order for application developers to use a framework correctly, they must learn how to invoke methods in the framework's API, which we call *API methods*, in order to implement application specific features as well as the design of the framework [2]. Usually such method invocations are tangled and are scattered in an application specific code, and thus it is difficult to identify which invocations implement a feature. A static analysis [3] and a dynamic analysis [4] are proposed in order to solve this problem.

These existing approaches suit well if maintainers don't have to understand what invoked methods do inside of their framework applications. However, in real software maintenance tasks, such a kind of program comprehension is often required even if a correct way of API method invocations are known [3], [2]. A new approach is necessary for understanding internal behaviors of framework applications. A proper abstraction is the key to success, we believe.

This paper proposes a behavioral model, called *feature component model*, which abstracts complex internal behavior in executing a particular feature. A feature component model represents behavioral dependencies among method invocations in terms of *method parameters*, which are method receivers and arguments, and runtime states accessed via objects.

In this paper we introduce a software maintenance case which we have resolved a side effect caused by a misuse of a practical framework developed by a third party. In our previous study[5] we have shown that the side effect can be thought as a *feature interaction*. A feature interaction is a situation that features each of which works correctly by itself cause an unexpected result by their combination [6], [7].

In the previous study, we were required to examine the source code of the framework application in order to resolve the side effect. Our feature component model originally aims at supporting such source code examination for framework applications. We will evaluate our approach by applying it to the source code examination task.

The rest of this paper is organized as follows: We first explain concepts and terms of framework applications in section II. Next we explain the example of framework misuse that results in the side effect in section III. In section IV we introduce our feature model and apply it to the misuse example. We evaluate our model in section V. We explain related work in section VI, and discuss our future work in section VII. Finally, we conclude in section VIII.

II. Preliminaries

An application framework is a software product which provides a reusable design and an implementation for applications in a particular domain [8]. An application built on a framework consists of the classes in the framework and the classes added to implement application specific features. We call the former classes *framework classes* and the latter *application specific classes*. We call the methods implemented in framework classes *framework methods*, and those implemented in application specific classes *application specific methods*.

A framework is thought to "provide domain-specific concepts in a particular domain, which are generic units of functionality" [2]. Framework classes that represent concepts of an application domain by API methods are called *the core classes* [10]. Core classes include so called *hot spots* from which framework concepts are extended by inheritance.

In general, a feature is defined as "a realized functional requirement of a system"[9]. *A feature* of a framework application is defined as an observable behavior which is implemented by extending framework concepts and by invoking framework API methods.

978-0-7695-5005-3/13 $26.00 © 2013 IEEE
DOI 10.1109/SNPD.2013.14

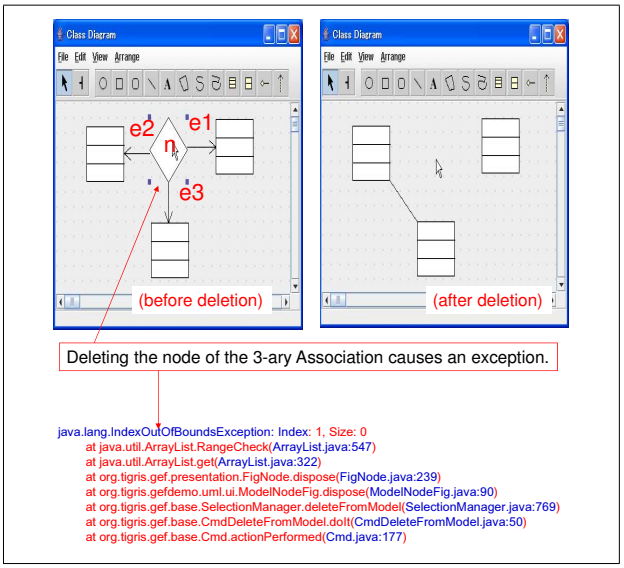

Fig. 1. Exception at Deleting N-Ary UML Association

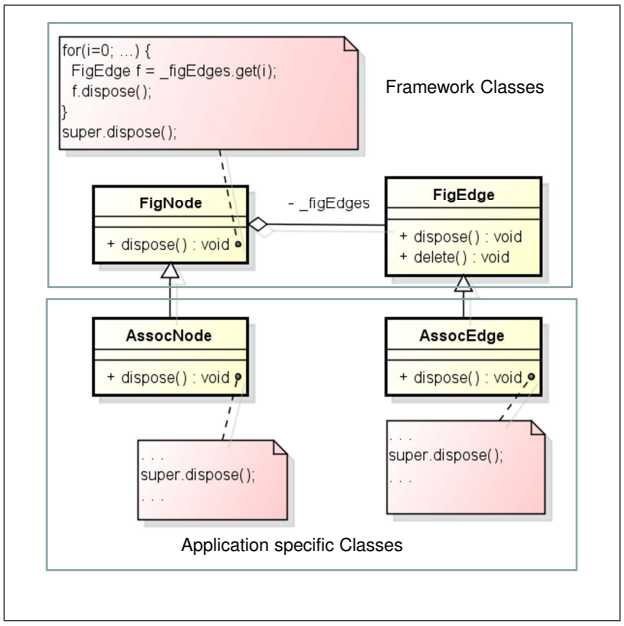

Fig. 2. Architecture of Framework Application

Application specific methods which override the methods of hot spots are lexically referenced as the overridden methods and are invoked in framework methods. Pree[11] named such invoking framework methods *template methods* and referenced methods *hook methods*. This invocation style is a part of so called *inversion of control* [12], an execution style specific to framework applications.

In building an application on a practical framework, application developers must implement method invocations of core classes correctly. Such invocations from the application specific part to the framework is necessary for various reasons including initializing some concepts, setting up of inversion of controls, or notification of state change on some objects under management of the framework, etc. In this paper, we call such invocations *re-inversion of controls*. Existing approaches [4], [3], [1] aims at learning how to implement re-inversion of controls from the source code of sample application in the absence of proper documentation. Our approach, on the other hand, aims at what such re-inversion of controls do in executing a feature.

III. Example of Framework Misuse

GEF(Graph Editing Framework)[1] is a practical framework for graph editors. GEF is deployed as an open source Java program with an example application, which is a UML diagram editor. GEF implements graph nodes and edges as general graph elements as well as their related concepts such as deletion of a graph node. The example application implements nodes and edges of UML n-ary Associations by extending the graph nodes and edges, respectively, as well as their related features such as deletion of a UML n-ary Association.

Figure 2 shows the architecture of this example application. Two hot spot classes `FigNode` and `FigEdge` are shown with the pseudo code of their methods at the upper side of this figure. At the lower side of this figure, two application specific classes `AssocNode` and `AssocEdge` [2], which inherits `FigNode` and `FigEdge` respectively, are shown. Framework classes `FigNode` and `FigEdge` represents nodes and edges of a general graph structure, respectively. Their application specific subclasses `AssocNode` and `AssocEdge` implements nodes and edges of n-ary UML Associations, respectively.

Methods named `dispose()` of `FigNode` and `FigEdge` implement concepts to delete a graph node and a graph edge respectively. Any application must invoke `dispose()` on the deleted element, if it tries to delete a graph node or a graph edge. Such invocation is necessary for the deleted element to disappear visually.

In the example application, the application specific classes `AssocNode` and `AssocEdge` override `dispose()` of their superclasses in order to implement deletion feature specific to UML Association nodes and edges. The overriding methods invoke their overridden `dispose()` by `super.dispose()` for the rule mentioned above. At runtime the overriding `dispose()` methods, which are referenced lexically by some template methods, are invoked as their hook methods. The invoked hook methods then invoke the overridden `dispose()`. As a result, deleting a n-ary UML Association node triggers an inversion of control and its succeeding re-inversion of control with respect to `dispose()`.

This application throws an exception when deleting a 3-ary UML Association. Figure 1 shows a user operation that triggers the exception throwing. Here, three UML Classes are connected by a 3-ary UML Association with one node (`n`) and

[1] http://gef.tigris.org/

[2] Their real names are too long and are abbreviated for the simplicity of our explanation.

three edges. An exception is thrown when the node is selected and DELETE key is pressed. The error message in figure 1 tells that there is an attempt to get an item from an empty list, which suggests a side effect on the list object.

In our previous study [5], we proposed a dynamic analysis method to represent the above side effect as a feature interaction between n-ary UML Association node deletion and general graph edge deletion. However, when we resolved the feature interaction, we had to examine the source code not only of the example application but also its framework GEF. In general, it is desirable for maintainers not to examine the implementation details of frameworks. Therefore, we pursue an approach that enables maintainers to understand internal behaviors of invoked methods and their effects on others without examining the implementation details of a framework.

IV. Feature Component Model

We propose a behavioral model called *feature component model* that abstracts dependency among method invocations in inversion/re-inversion of controls in terms of method parameters of invocations and runtime states changed by invoked methods. Given a behavior of an inversion/re-inversion of control, say **C** our feature component model aims at help maintainers understand which inversion/re-inversion of controls make **C** behave so.

As for our example in section III, we expect that its feature component model shows which re-inversion of control triggers the side effects according to a runtime state accessed through an object passed as a method parameter. Maintainers should obtain this observation without examining the implementation details of the GEF framework.

For maintainers to obtain such a kind of observation using a feature component models, we assume that concepts and features have been located by existing *feature location techniques* (e.g. [2], [13]). In our example, we assume that maintainers understand the concepts and features implemented by `FigNode`, `FigEdge`, `AssocNode` and `AssocEdge` as we explained in section III.

A. Features and Feature Components

As we have seen in our example in section III, application specific methods are often required to invoke API methods of its framework. Such invocations are usually executed under some inversion of controls. As a result, we often see nested inversion/re-inversion of controls in a framework application at runtime.

As for our example, figure 3 shows nested inversion/re-inversion of controls in the process of attempting to delete the 3-ary UML Association node in figure 1. This figure shows under the method invocation tree (1) an exception throwing statement and (2) an update statement to make a list empty by assignment of `0` to instance variable `size`. These statements are surrounded by dotted yellow lines and assigned their unique ID numbers (`0` and `8`).

Each of the invoked methods is assigned an alphabet and a unique ID number. Alphabets `F`, `A`, and `J` represent a framework method, an application specific method, and a

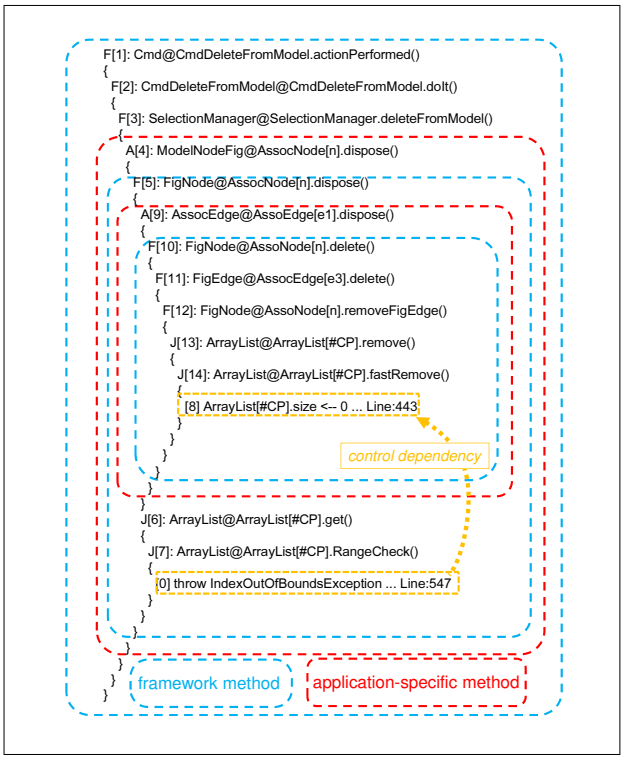

Fig. 3. Invocation Context of Exception Throwing

Java library method[3] respectively. Method invocation trees in an inversion of control and a re-inversion of control are surrounded by dotted lines in red and blue respectively.

We call a method that implements a feature or its concept, and that is invoked at an inversion/re-inversion of control *feature method*. Methods in an invocation tree are clustered according to their belongings. Each invocation cluster, which is called *feature component* has a feature method or a Java library method at its top. Thus a method invocation tree is abstracted as an invocation tree of runtime feature components, which depicts inversion/re-inversion of controls in an execution.

In general, a runtime feature component contains many statements such as conditional branch instructions and exception throwing statements. It also contains method invocation trees under its feature method. Note that figure 3 contains the exception throwing statement inside of the Java library class `ArrayList` but not the conditional branch instruction which decides to throw the exception.

All values referenced in a method execution are (1) created inside of the method, (2) passed from another methods as receivers, arguments , (3) obtained from objects as their instance values, array components or return value of method invocations on them, or (4) calculated or derived (e.g. via Java

[3]In order to avoid possible license problem, the bytecode of `ArrayList` is replaced by that of another class with a different package name. Its implementation is very similar to `ArrayList`. We obtained the substitute class from Open JDK.

tt instanceof operation) using other values[4].

We call the above categorized values *internal values*, *method parameters*, *carried values*, and *derived values*, respectively. For a given derived value, we can reach a set of internal values, method parameters, and carried values by going back on its derivation process. A carried value can be obtained via a class instance or an array, which we call *carrier*.

Carriers themselves are obtained as internal values, as method parameters, or even as carried values. As a result, all referenced objects are internal values or method parameters, otherwise are obtained via these values. For a referenced value, the method parameters from which the value is obtained are called `representative parameters`. Note that an object value has only one representative parameter.

B. Dependency Abstraction

A feature component model introduces three abstraction levels of an execution of a framework application. First, method invocations in a runtime feature component are abstracted out. Second, statements in a runtime feature components are abstracted out except for exception throwing, assignment statements of instance variables and array components, and conditional branches. Third, values referenced in a runtime feature component are represented by their representative parameters. Internal values are simply abstracted out.

As for instance variables and array components accessed in a runtime feature component. their owners (class instances and arrays) may be represented by their representative parameters. The same abstraction is applied to receivers of method invocations with return values. According to this abstraction about object values, assignment and reference statements are represented as internal state updates via their representative, and value obtainment from internal state via representative parameters, respectively.

A feature component model represents a dependency relationship among runtime feature components. This dependency can be derived from lower level dependency among statements and values referenced by these statements. This lower level dependency has an aspect similar to data/control dependency [14] and *aliases* to objects [17].

C. Example Feature Model

Figure 4 shows diagrams[5] that illustrate the runtime feature components under `F[5]` in figure 3, and the dependency among the runtime feature components in different levels of details. Here, object `n` is passed to `F[5]`, the root of the whole invocation, as its only parameter. It is the instance of `AssocNode`, whose deletion results in the exception throwing in the example of section III.

The upper diagram shows the effect of application specific feature component `A[9]` which makes the exception thrown. Actually `A[9]` invokes framework feature component

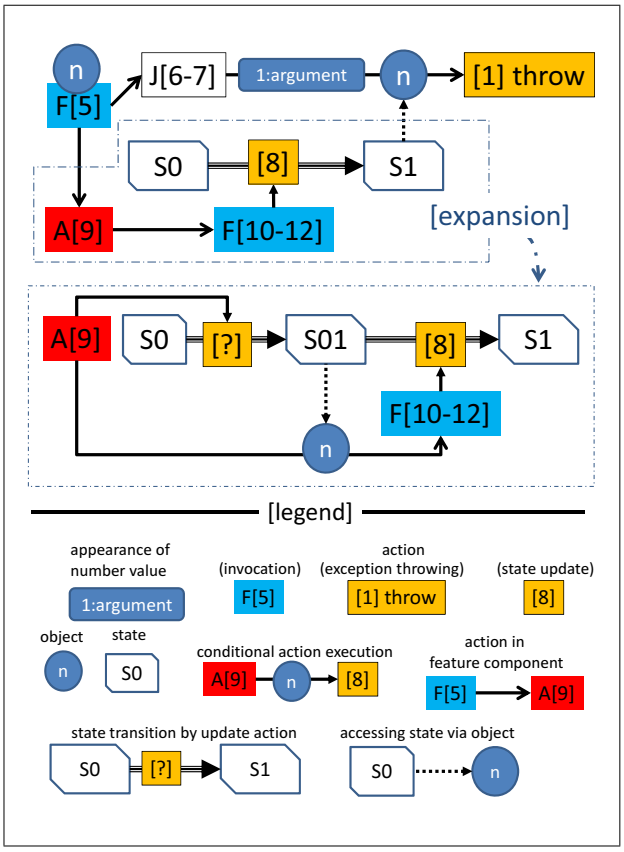

Fig. 4. Example Feature Model

`F[10-12]`, which makes an internal state transition from `S0` to `S1`. Invocation of Java library method `J[6-7]` obtains a value from the new state `S1` via `n`. The state change was caused by runtime feature component `F[10-12]` invoked by `A[9]`. As a result, the application specific method `A[9]` makes the side effects in the internal state via the invocation of `F[10-12]`.

The lower diagram in figure 4 and the diagram in figure 5 illustrate the inside details of `A[9]` and `F[10-12]`, respectively. The former diagram shows that `A[9]` does some internal state update from `S0` to `S01` via `n`. Then it decides to invoke `F[10-12]` based on the result of its own update, which is obtained via `n`. The diagram in figure 5 shows that `F[10-12]` decides internal state update from `S01` to `S1` based on the current state `S01`, which has been set by `A[9]`.

The dependency derivation among runtime feature components are supported by the rich data types of program traces generated our trace analysis tool[5]. A generated trace records not only method invocations but also their statements. It also records all number values and objects with their appearances in the recorded statements as well as their appearances as method parameters or return values. Therefore, our tool covers not only aliases to objects [17] but also references to number values. A trace records the relationship among an statement and the conditional branch that executes it if any.

[4]For the simplicity of our discussion, we don't deal with class variable values.

[5]At this stage, we made these diagrams manually with the aid of a trace analysis tool we have developed[5].

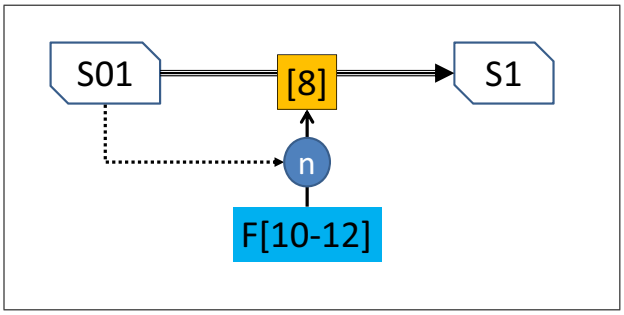

Fig. 5. Inside `F[10-12]`

Given a value (either a number value or an object) referenced by a conditional branch, our tool can show the reference record from its creation to the current reference. From the reference record of the value, we can detect all statements that participate in the process to bring the value to the conditional branch. In other words, we can obtain a sequence of statements that makes the decision by the conditional branch. We call such statements `the decision makers` of the conditional branch.

A dependency relationship among runtime feature components is derived from the conditional branches and their decision makers, while the conditional branches are grouped and most of their decision makers are abstracted out.

At the construction of a feature component model, maintainers are required to select one or more feature method invocations as the subject of the model construction. By default, the method parameters of the selected top-level invocations are specified as the representative parameters of the constructed model.

V. Evaluation

Our feature component model well abstracts complex execution of framework applications in terms of feature components and their dependency relationship. We will evaluate the case study of our approach explained in section IV-C from the viewpoints of its simplicity and usefulness.

As we saw in figure 4 and figure 5 the complex invocation structure under `F[5]` (figure 3) are well simplified as three two framework runtime feature components (`F[5]` and `F[10-12]`), one application specific runtime feature component (`A[9]`), and an invocation tree of Java library method `J[6-7]`.

The whole invocation tree of `F[5]` contains so many actions and complex control structure. In addition to them, number values and objects are references. These objects are themselves work as carriers and/or carried values, resulting in too complex control/data dependency. The feature component model contains only one object (the `AssocNode` instance `n`) and a number value `1`, which appear as the method parameters of `F[5]`. Only three grouped conditional branches and two abstracted forms of state updates are contained.

The simplicity about the dependency among runtime feature components is partially due to the selection of the actions and value references which participate in the exception throwing (denoted as `[0]`). Grouping of conditional branches and value representation by representative parameters contribute crucially to the simplicity.

The feature component model of our example illustrates the dependency among the feature components at a glance. More precisely to say, it illustrates which values and internal states the feature components relies on for their decision at runtime. It also illustrates who made them their decision via the referenced values and states. In this way, maintainers can grasp 'Who makes the decision?' at a glance.

Let's examine the internal behavior of `J[6-7]` which decides to throw the exception. As for this decision, a number value `1` and a state `S1` is referenced. The former is created internally in `F[5]` [6] and passed to `J[6-7]` as an invocation argument. This internal creation is decided by `F[5]` itself. Therefore, if we trust the GEF, we should trust the argument value.

As for the state `S1`, it is set by framework feature component `F[10-12]` directly. However, as we can see from the lower diagram in figure 4 and another in figure 5, the decision was made by application specific feature component `A[9]` via its state change from `S0` to `S01`. The decision by `F[10-12]` is solely due to this state change, and thus `A[9]` is responsible for the state change by `F[10-12]`.

The above observation makes a good suggestion about where maintainers should start their source code examination. Clearly, they should start at `A[9]`, method `dispose()` of `AssocNode` which override the framework method defined in `FigNode`. As we explain precisely in our previous work [5], it is actually a useful starting point for efficient resolution of a framework misuse that causes the side effect.

Although our case study belongs to a framework misuse problem [5], the most important point for its solution is the support for program understanding about the internal behavior of the framework application. Such a kind of program understanding about internal behaviors hidden under method invocations is beyond the scope of existing approaches [2], [4], [3] which put their focus on analysis on method invocations with related to re-inversion of controls.

VI. Related Work

Tyler and Soundarajan propose a test method to check the timing of inversion of control, which is abstracted out in the code of frameworks[15]. Heydarnoori et al. [4] propose a dynamic analysis approach to construct a 'template' of framework method invocations to implement a framework concept. They evaluate their approach by the usefulness of constructed templates in program understanding [2].

Monperrus et al. [3] propose a combination of static analysis and statistics in order to cope with the 'missing method call' problem, a kind of framework misuse where method invocation necessary for correct re-inversion of control is missed. Their method deal with only method invocations from an application specific part to its framework only. They noted the importance

[6]Note that its internal creation is abstracted out.

of program understanding in a maintenance task which is beyond the scope of their approach.

Apart from framework applications, dependency in object-oriented programs is a main subject of many dynamic analysis approaches [16], [17], [18]. The analysis target of the approach by Salah and Mancoridis [16] is the dependency derived from object creation and reference. Dynamic object flow analysis by Lienhard and et al. [17] extend the dependency including object references as method parameters and instance variable values. The approach by Wang and Roychoudhury [18] uses data dependency to divide a long trace into phases.

A feature interaction is a situation that a combination of features, each of which works correctly, results in an unexpected result. Feature interactions are originally studied in the area of telecommunication or Web applications systems [6], [7]. The side effect example in section III can be thought as a feature interaction between a feature of node deletion and an extended feature of edge deletion [6], [7].

VII. Future Work

At this stage, the process of dependency abstraction of a feature component model includes a partially manual task. We have not implemented visualization of a constructed model as diagrams. Maintainers' interaction in model construction is necessary for selecting method invocation trees under analysis, representative parameters, and etc. We implement the lacking functions on the trace analysis tool we have developed [5]. At the same time, we will develop a general method to use the tool for efficiently constructing models which are easy to understand, and provide useful information for maintainers.

VIII. Conclusion

In this paper we proposed a behavioral model, called *feature component model*, that aims at supporting framework users. Our feature component model simplifies a complex execution of a framework application in terms of *feature components*, which are abstraction of inversion/re-inversion of controls, and their mutual dependency. We demonstrated the usefulness of our approach by applying it to a case study of resolving a side effect, which normally requires time and effort.

Acknowledgments

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), Scientific Research (B) (No.23300009)], and Kansai Research Foundation for technology promotion.

References

[1] F. Shull, F. Lanubile, and V. R. Basili, "Investigating reading techniques for object-oriented framework learning," *IEEE Transactions on Software Engineering*, vol. 26, no. 11, pp. 1101–1118, 2000.

[2] A. Heydarnoori, K. Czarnecki, W. Binder, and T. T. Bartolomei, "Two studies of framework-usage templates extracted from dynamic traces," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1464–1487, 2012.

[3] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *ECOOP*, 2010, pp. 2–25.

[4] A. Heydarnoori, K. Czarnecki, and T. T. Bartolomei, "Supporting framework use via automatically extracted concept-implementation templates," in *ECOOP*, ser. LNCS 5653. Springer-Verlag, 2009, pp. 344–368.

[5] I. Kume, M. Nakamura, and E. Shibayama, "Toward comprehension of side effects in framework applications as feature interactions," in *he 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, 2012.

[6] M. Nakamura, H. Igaki, and K. ichi Matumoto, "Feature interactions in inegrated services of networked home appliances," in *Feature Interactions in Telecommunications and Software Systems*, 2005, pp. 236–251.

[7] 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.

[8] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, June/July 1988.

[9] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, Match 2003.

[10] M. E. Fayad, D. C. Schmidt, and R. E. Johnson, *Building Application Frameworks*. John Wiley & Sons., 1999.

[11] W. Pree, *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[12] S. Sparks, K. Benner, and C. Faris, "Managing object-oriented framework reuse," *IEEE Computer*, vol. 29, no. 9, pp. 52–61, 1996.

[13] T. Eisenbarth, R. Koschke, and D. Simon, "Feature-driven program understanding using concept analysis of execution traces," in *International Workshop on Program Comprehension*. IEEE, 2001, pp. 300–309.

[14] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[15] B. Tyler and N. Soundarajan, "Testing framework components," in *Component-Based Software Engineering*, ser. LNCS 3054. Springer, 2004, pp. 138–145.

[16] M. Salah and S. Mancoridis, "A hierarchy of dynamic software views: From object-interactions to feature-interactions," in *International Conference on Software Maintenance*. IEEE, 2004, pp. 72–81.

[17] A. Lienhard, *Dynamic Object Flow Analysis*. Lulu.com, 2008.

[18] T. Wang and A. Roychoudhury, "Hierarchical dynamic slicing," in *International Symposium on Software Testing and Analysis*. ACM, 2007, pp. 228–238.

[19] S. Uchitel, R. Chatley, J. Kramer, and J. Magee, "Ltsa-msc: tool support for behaviour model elaboration using implied scenarios," in *TACAS'03 Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*. Springer-Verlag, 2003, pp. 597–601.