Detecting Invalid Layer Combinations Using Control-Flow Analysis for Android

Noriyuki Suzuki
Ritsumeikan University, Japan
noriyukisuzuki@fse. cs.ritsumei.ac.jp

Tetsuo Kamina
Ritsumeikan University, Japan
kamina@acm.org

Katsuhisa Maruyama
Ritsumeikan University, Japan
maru@cs.ritsumei.ac.jp

Abstract

In Context-Oriented Programming (COP), it is possible that *invalid combinations between layers* (a set of activated layers that violates some required properties) occur at runtime. Even though such combinations can be detected using runtime checking, it potentially requires a significant amount of cost for testing. In this paper, we propose a method to detect invalid combinations between layers using state-of-the-art control-flow analysis for Android applications. Using Android specific knowledge, such as the layout file for GUI components, reasonably precise callback sequences in Android applications are actually constructed, and our method applies this fact to the analysis of COP programs. Using a simple example, we demonstrate how our method finds invalid combinations between layers.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Validation

General Terms Algorithms, Languages

Keywords Layer activation coverage, Callback control-flow graph, Event-based COP

1. Introduction

Context-Oriented Programming (COP) is an approach to improve the modularity of behavioral variations that depend on contexts. A number of COP languages provide linguistic constructs that modularize such variations using *layers* and dynamically activate/deactivate them according to the executing contexts (6; 8). A layer defines *partial methods*, which run before, after, or around a call of a method with the same signature defined in a class only when the layer is activated. These constructs give advantages to COP in modularity, because partial methods can change the original behavior by activating layers without changing the base classes.

However, several COP languages activate layers independently, which may cause invalid combinations between layers (i.e., a set of layers that violate some required properties). Even though several COP languages activate layers in disciplined manners, e.g., using scoping (6) or composite layers (5; 11), such activation mechanisms do not prohibit layer activation that is mistakenly specified by the programmer. Even though such invalid combinations between layers can be detected using runtime checking (7), it potentially requires a significant amount of cost for testing.

In this paper, we propose a method to detect invalid combinations between layers that occur at runtime using static analysis. Our method is based on control-flow analysis. We first construct a control-flow graph (CFG) of the given application. Then, for each layer, we identify a set of nodes of that CFG where that layer is assumed to be activated using a simple depth-first search (DFS) algorithm, and check whether each set satisfies the properties of layers given by the programmer. As properties, the programmer can specify the *alternative* relation (i.e., at most one layer from the specified set of layers can be activated at the same time) and the *requires* relation (i.e., activation of a specified layer requires that of other layers). We note that, in this paper, we focus on event-based COP (10) as well as control-flow-based one (8). Other activation mechanisms such as using conditionals (14) are considered beyond the scope of this paper.

For constructing the CFG, we make a *callback control-flow graph* (CCFG) (15), because a number of COP applications use user-driven callbacks, which make the precise control-flow graph construction very difficult. To address this problem, CCFG uses Android specific knowledge, i.e., the XML layout file equipped with the Android application to directly connect the actions that trigger events in the user code with the corresponding event handlers.

Using a simple example, we demonstrate how our method finds invalid layer combinations. We tested our method on four cases, where, in two cases, we intentionally specified invalid layer activation, and two other cases were the correct ones. We found one false positive; in other cases, our

[Copyright notice will appear here once 'preprint' option is removed.]

```
public class Nav {
  public void getPosition() {
    System.err.println(".."); }
}
layer Outdoors {
  public class Nav {
    public void getPosition() {
      // GPS based positioning
      Location loc = ... ;
      ...  }
  } }
layer Indoors {
  public class Nav {
    public void getPosition() {
      // Floor-specific positioning
      Location loc = ... ;
      ...  }
  } }
```

Figure 1. `Outdoors` and `Indoors` layers

method successfully detected the invalid layer combinations. The false positive occurred when conditions for layer activations were hardwired within the base program, which might imply that the program should be refactored. Even though this work is still in its early stage, we believe that this research direction, i.e., using CFG for Android applications to analyze layer activations, is promising.

2. Problem of Independent Layer Activation

Example. We explain our motivation using a simple pedestrian navigation system that provides behavioral variations of positioning mechanisms and map views for both outdoor and indoor situations. Those variations, which can be switched dynamically, are implemented using layers. Figure 1 illustrates those layers, namely, `Outdoors` and `Indoors`, that change the behavior of the `getPosition` method in the `Nav` class to provide context specific positioning mechanisms (i.e., a GPS-based positioning for the outdoor situation and a floor-specific positioning for the indoor situation) when the corresponding layer is activated. The map view also changes with respect to situations (i.e., a city map for the outdoor situation and a floor plan for the indoor situation), which is not shown for brevity.

We also consider an additional feature, namely, a satellite view, which can be used only when it is in the outdoor situation. The satellite view provides a satellite image of the city map, and it is also implemented using a layer, namely, `Satellite`.

Those layers are activated and deactivated dynamically. To ensure system consistency, there are several requirements for layer activation.

- When `Outdoors` is activated, it is assumed that `Indoors` is not activated, and vice versa. This is because both provide completely alternative behaviors.

```
contextgroup NavGroup() {
  activate Outdoors
    from Nav.startOutdoors to Nav.startIndoors;
  activate Indoors
    from Nav.startIndoors to Nav.startOutdoors;
  activate Satellite
    from Satellite.switch to Satellite.switch
    && when isActive(Outdoors); }
```

Figure 2. Layer activation in ServalCJ

```
contextgroup NavGroup() {
  ..
  activate Indoors
    // specifying different event
    from Nav.startOutdoors to Nav.startOutdoors;
  activate Satellite
    // forgetting "when isActive(Outdoors)"
    from Satellite.switch to Satellite.switch; }
```

Figure 3. Mistakenly specified layer activation

- The activation of `Satellite` assumes that the activation of `Outdoors`, because `Satellite` depends on the behavior provided by `Outdoors` such as the GPS-based positioning.

Layers can be activated by specifying dynamic extent (6; 8), generating events (10), and declaratively specifying conditions when those layers are activated (14). Figure 2 describes layer activation in ServalCJ (12), which generalizes those layer activation mechanisms in one single language. Layers `Outdoors` and `Indoors` are activated using events specified in the `from` clause, and those are activated until other events (specified in the `to` clause) are generated. The name of the event is prefixed by the class name where that event is declared. In ServalCJ, each event can be generated using a syntax equivalent to a method call, e.g., `startOutdoors()`. Each event generation is semantically equivalent to the corresponding join point in AspectJ. Thus, when the program execution reaches the join point where `startOutdoors` is fired, then `Outdoors` is activated. The activation of `Satellite` is specified using the composite layer mechanism (11); i.e., it is activated when `switch` is fired *and* `Outdoors` is activated. If events specified in `from` and `to` clauses are the same, then this means that the activation happens from the first event until the second event.

Problem. Generally, there are required properties (constraints) between layers; e.g., some layers provide conflicting behaviors, and thus they cannot be activated at the same time; or, some layers provide behaviors that make sense only if some other layers are activated. For example, `Outdoors` and `Indoors` provide conflicting behaviors, and `Satellite` is executable only when `Outdoors` is activated.

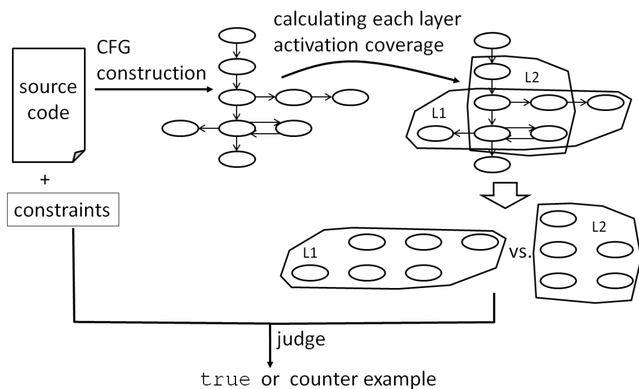

Figure 4. Overview of our method

A problem arises when layers are activated in a way that violates such constraints. A programmer may mistakenly specify the faulty layer activation. For example, the above constraints are violated if the programmer specifies the layer activation as shown in Figure 3. We call a set of activated layers that violates the constraints an *invalid combination of layers*. The specifications in Figure 3 result in invalid combinations, e.g., {`Indoors`, `Outdoors`}, which cause unexpected behaviors like that the indoors behavior overrides the outdoors one when the user is in the outdoor situation.

To eliminate the invalid combinations, a mechanism for detecting them is required. Even though they can be detected using runtime checking (7), it requires possibly a significant amount of cost for testing.

3. Detecting Invalid Combinations

We propose a method to detect invalid combinations between layers that occur at runtime using static analysis. In particular, we apply an analysis method for control-flows with callbacks in Android applications (15) to detect invalid combinations between layers.

Overview. Figure 4 summarizes our method. Our method takes the source code of the ServalCJ program as an input, as well as the constraints between layers provided by the programmer. It constructs a CFG of the base program, and, based on the layer activation specifications provided by the `contextgroup` declarations, it calculates the set of nodes in CFG on which the specified layer is activated, which we call a *layer activation coverage*. By comparing each layer activation coverage, our method decides whether the constraints are satisfied.

Specifying constraints. Constraints in our method are based on *feature diagrams* (13). A feature diagram represents a hierarchically arranged relations between features of a particular domain; i.e., this is a tree where primitive features are leaves and compound features are interior nodes. Relationships between a parent feature and its child features include *mandatory* (features that required), *optional* (features that are optional), *alternative* (only one child feature

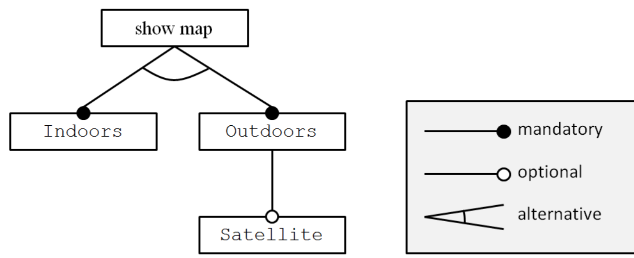

Figure 5. A feature diagram

can be selected), and so on. A feature diagram is a widely adopted discipline to represent product's commonality and variability, and it is a handy way to represent the dependencies between layers.

In our method, we represent constraints between layers using a feature diagram. Figure 5 shows an example. The root feature "show map" is a domain that this diagram characterizes. All layers are child features of particular domains (or child features of some other layers). For example, `Indoors` and `Outdoors` are child features of "show map" and these are in the alternative relationship. `Satellite` is a child feature of `Outdoors` and the former can be selected only if the latter is selected, while the latter does not always require that the former to be selected. These constraints are written in the following textual format.

```
@alternative(Outdoors,Indoors)
@requires(Satellite->Outdoors)
```

Constructing CFG. The ServalCJ program is checked against the constraints provided by the programmer by statically estimating activated layers for each statement of the program. To perform such a check, a model representing all the possible computations is necessary. A control-flow graph (CFG) is one of such models where all possible control-flows of the execution of the given program are represented, and our method is based on the interprocedural CFG (ICFG), where CFGs of the program's procedures are combined.

At first, the traditional ICFG looks unsuitable for our purpose. Traditional control-flow analysis is unsuitable for framework-based and event-driven applications where the application code and the framework interact through callbacks, because there are no abstractions for handlers that handle particular events. Context-aware applications (and thus applications written in COP languages) are inherently event-driven because they reactively respond to environmental changes, which are usually represented as events.

Instead of the traditional ICFG, we apply the *callback CFG (CCFG)* (15) to address this issue. A CCFG is a variant of CFG that captures the callback sequences in Android applications. By analyzing the XML layout file equipped with the Android application and relevant code, in CCFG, event handlers are directly related to actions in the user code that may trigger events handled by the event handlers.

Figure 6 shows a CCFG of the example program shown in Figure 7. This graph shows sequences of lifecycle callbacks

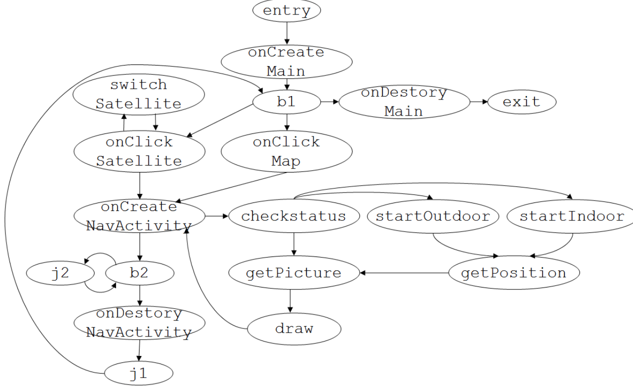

Figure 6. CCFG for the example program

```
class Map .. {
  public void onClick(View v) {
    switch(v) {
    case nav:
      ../* Trantition to NavActivity */
    case satellite:
      ../* Transition to Satellite */
    }
  } }
class NavActivity .. {
  public void onCreate(..) {
    checkStatus();
    getPosition();
    getPicture();
    draw();  }
  void checkStatus() {
    if (..) { startOutdoors(); }
    else if (..) { startIndoors(); }
  } }
class Satellite .. {
  public void onClick(..) {
    switch();
    ../* Transition to NavActivity */
  } }
```

Figure 7. An example base program. We note that `startOutdoors()`, `startIndoors()`, and `switch()` are not method invocations but event generations in ServalCJ.

(such as `onCreate` and `onDestroy`) and GUI event handler callbacks (such as `onClick`), as well as CFGs within each event handler. Branch nodes b_i and join nodes j_i indicate that event handlers can be executed in any order. We note that, in Figure 6, event generations in ServalCJ such as `startOutdoors`, `startIndoors`, and `switch` are also represented as CCFG nodes.

Deciding layer activation coverage over CFG. A layer activation coverage is calculated by first selecting nodes in CCFG that activate and deactivate a particular layer. This is simply performed by selecting nodes that match events

Procedure: $DFS(n_{start}, n_{end})$
Input: n_{start} (activation node), n_{end} (deactivation node)
Output: R (all nodes visited from n_{start} to n_{end})

1: $R \leftarrow \{n_{start}\}$
2: $E \leftarrow$ set of outgoing edges from n_{start}
3: **for all** $e_i \in E$ **do**
4: $\quad n' \leftarrow$ sink of e_i
5: $\quad$ **if** $n' \notin R$ and $n' \neq n_{end}$ **then** $DFS(n', n_{end})$
6: **end for**

Figure 8. Layer activation coverage in CFG

specified in the `from` and `to` clauses in the `activate` declarations for the corresponding layer. For example, in Figure 6, we found that `Outdoors` is activated in the node labeled `startOutdoors` and deactivated in the node labeled `startIndoors`.

After selecting the layer (de)activation nodes, we apply a simple DFS algorithm from the activation node to the deactivation node. This algorithm is shown in Figure 8. In short, it returns all nodes that may be visited during the execution from n_{start} to n_{end}. Note that, if there are multiple activation and deactivation nodes, all possible combinations between them are considered. For example, the layer activation coverage of `Outdoors` in Figure 6 results in `startOutdoors`, `getPosition`, `getPicture`, `draw`, `onCreate` (in `NavActivity`), `b2`, `j2`, `onDestroy` (in `NavActivity`), `j1`, `b1`, `onClick` (in `Satellite`), `switch` (in `Satellite`), `onClick` (in `Map`), and `checkStatus` (note that $n_{\mathtt{end}}$ is not included in the result).

The above algorithm explains the case where an event-based activation is used. We note that this algorithm is also applied to the control-flow-based activation (e.g., `cflow` in ServalCJ and `with` in ContextJ (1)). Actually, in our method, the control-flow-based activation is considered as a special case of the event-based activation where the activation and deactivation nodes are the same.

Checking invalid combinations. Based on the layer activation coverage, we check that the constraints provided by the programmer are satisfied. We provide the different algorithms for checking two kinds of constraints: `alternative` and `requires`.

The algorithm for `alternative` is shown in Figure 9. We write the set of all activation nodes of layer A as N_{A_start}, and the set of all deactivation nodes of layer A as N_{A_end}. When *DFS* is applied to sets S and T, it is interpreted as $DFS(S,T) = \bigcup_{s_i \in S, t_j \in T} DFS(s_i, t_j)$. Intuitively, $checkAlternative(A, B)$ checks that no activation nodes of B are contained in the layer activation coverage of A, and no activation nodes of A are contained in the layer activation coverage of B.

For example, *checkAlternative*(`Outdoors`, `Indoors`) returns `true` if layer activations are specified as Figure 2, but returns `false` if these are specified as Figure 3 be-

Procedure: *checkAlternative*(A, B)
Input: A (layer), B (layer)
Output: `true` if A and B satisfy the alternative constraint; `false` otherwise

1: $R_A \leftarrow DFS(N_{A_start}, N_{A_end})$
2: $R_B \leftarrow DFS(N_{B_start}, N_{B_end})$
3: **for all** $R_i \in R_A$ **do**
4:     **for all** $n \in R_i$ **do**
5:         **if** $n \in N_{B_start}$ **then return** `false`
6:     **end for**
7: **end for**
8: **for all** $R_i \in R_B$ **do**
9:     **for all** $n \in R_i$ **do**
10:         **if** $n \in N_{A_start}$ **then return** `false`
11:     **end for**
12: **end for**
13: **return** `true`

Figure 9. Checking alternative constraint

cause the layer activation coverage of `Indoors` contains `startOutdoors`, which is the activation node of `Outdoors`.

We omit the algorithm checking the `requires` relation in this paper because it is very simple: it simply checks whether the layer activation coverage of the requiring layer is a subset of the coverage of the required layer.

Limitations. There are limitations to our method. First, our method assumes Android applications and other platforms are considered beyond the scope, because the CCFG construction mechanism is specialized to Android applications. Further research on constructing sequences of callbacks for other platforms (this construction inherently requires platform specific knowledge) would broaden the applicability of our method. Nevertheless, we consider that Android would be one of the major platforms for COP applications, and our method provides a good solution for statically detecting invalid combinations between layers that may occur at runtime. Second, our method assumes that all layer (de)activation nodes exist in the base program. To analyze the case where layer (de)activation occurs in the execution of some layer, a CFG construction mechanism that considers layers is necessary, which remains as future work. Third, our method focuses on event-based COP as well as control-flow-based one, and other activation mechanisms such as using conditionals are considered beyond the scope of this paper.

4. Preliminary results

We show how our method finds invalid layer combinations using the pedestrian navigation example. We prepared four cases based on the program shown in Figure 7; two were correct and the other two contained faults. Those cases are summarized as follows.

Case 1: Layer activation is specified as shown in Figure 2, where the `alternative` requirement between `Outdoors` and `Indoors` is satisfied.

Case 2: Layer activation is specified as shown in Figure 3, which violates the `alternative` requirement between `Outdoors` and `Indoors`.

Case 3: The same as the case 2, which also violates the `requires` requirement between `Outdoors` and `Satellite`.

Case 4: The activation of `Satellite` is specified as shown in Figure 3. A conditional branch is inserted before generating the `switch` event in Figure 7 to satisfy the `requires` requirement between `Outdoors` and `Satellite`.

We checked cases 1 and 2 against the property `@alternative(Outdoors,Indoors)` and cases 3 and 4 against the property `@requires(Satellite->Outdoors)`. In cases 2 and 3, our method effectively found the violations of the requirements. In case 1, our method did not find any requirement violations, which was also an expected result. However, in case 4, our method reported a requirement violation, even though this case did not violate the requirement. This is because our method conservatively draws edges for both following statements from one conditional branch in CFG without considering the actual values in the condition. This result implies that our method may not work correctly if conditions for layer activations are hardwired within the base program. This problem would be addressed by providing more advanced analyses for CFG construction (e.g., constant propagation or symbolic evaluation of conditionals), or simply refactoring the program to separately specify the conditionals in the `contextgroup` (e.g., the `isActive` specification in Figure 2 immediately leads to the satisfaction of the `requires` requirement).

5. Related work

There are several research efforts that use feature diagrams for COP applications. Costanza et al. proposed a method to analyze the dependency between layers using feature diagrams (5). As in our method, each feature corresponds to a layer. They also propose an extension of ContextL (6) with composite layers (that correspond to composite features), which provides layer composition operators that are as expressive as compositions in feature diagrams and an implicit layer activation mechanism based on those operators. Cardozo et al. proposed the feature clouds programming model (4), which also equates layers (i.e., "behavioral adaptations") to features in feature-oriented software development. This model clarifies correspondence between features and COP and advances the feature model by introducing dynamic adaptation of features by means of COP mechanisms. Our method complements those efforts by providing the static analysis to ensure the required properties defined between layers.

EventCJ (10) is a COP language that provides an event-based layer activation mechanism. For static analysis, it can generate a state transition specification from the program that can be checked by the model checker SPIN (9). Based

on SPIN, we can use linear temporal logic formula, where we can specify temporal properties that cannot be represented in the feature diagrams, to specify the required properties between layers. However, in EventCJ, the model of event generation is specified by the programmer, and the preciseness of the model is undertaken by the programmer. Instead, our method automatically determines the event generations from the CFG.

Some approaches also address the invalid layer combinations problem by allowing the declaration of dependencies between layers. Such approaches detect invalid layer combinations dynamically (2; 7). One notable exception is a Petri-net based formalism for context-oriented systems (3), where such combinations can be detected statically as well as dynamically. This formal model is used to reason about contexts and their interactions at design time. Instead, our method is applied at implementation time to analyze the source code to detect invalid layer combinations.

6. Concluding remarks and future work

This paper has proposed a method to detect invalid combinations between layers using static analysis. Using the state-of-the-art control-flow analysis for Android applications, it has been shown that the effective callback sequences in Android applications including events that activate and deactivate layers can be constructed, and our method can find the invalid combinations between layers using this analysis effectively. The preliminary study also revealed that there was a false positive that occurred when a condition for the layer activation are hardwired within the base program, and this case implies that the program should be refactored. Even though this work is still in its early stage, we believe that using CFG for Android applications to analyze layer activations is a promising approach.

There are a couple of research directions for future work. First, we are planning to refine our method to provide more precise analysis. One possible method is to develop a more precise estimation of the layer activation coverage. For example, instead of simple DFS, applying more precise analyses such as constant propagation and symbolic execution would produce more effective results. Another possible method to enhance the preciseness of analysis is to develop a layer activation aware CFG construction mechanism.

Another future research direction is enhancing the expressiveness of the constraint specifications. Currently our method supports only two specification constructs, namely, `alternative` and `requires`. It will be convenient if our method can support other dependencies such as implication or other dependencies that have been proposed by other method (7). Even though some of them may be represented by combining `alternative` and `requires`, it is interesting to explore new analyses on CCFG to represent more expressive dependencies that are not covered by those two.

Acknowledgments. This work was partially sponsored by the Grant-in-Aid for Scientific Research (15H02685) from the Japan Society for the Promotion of Science (JSPS).

References

[1] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, 2011.

[2] Nicolás Cardozo, Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. Run-time validation of behavioral adaptations. In *COP'14*, 2014.

[3] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, Jorge Vallejos, and Theo D'Hondt. Semantics for consistent activation in context-oriented systems. *Information and Software Technology*, 58:71–94, 2015.

[4] Nicolás Cardozo, Wolfgang De Meuter, Kim Mens, and Sebastián González. Features on demand. In *VaMoS'14*, 2014.

[5] Pascal Costanza and Theo D'Hondt. Feature descriptions for context-oriented programming. In *DSPL'08*, 2008.

[6] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *DLS'05*, pages 1–10, 2005.

[7] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In *SLE'11*, volume 6563 of *LNCS*, pages 246–265, 2011.

[8] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[9] Gerard J. Holzmann. The model checker SPIN. *IEEE Transaction on Software Engineering*, 23(5):279–295, 1997.

[10] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264, 2011.

[11] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Introducing composite layers in EventCJ. *IPSJ Transactions on Programming*, 6(1):1–8, 2013.

[12] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *MODULARITY'15*, pages 14–28, 2015.

[13] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[14] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *ICDL'07*, pages 143–156, 2007.

[15] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *ICSE'15*, pages 89–99, 2015.