Push-based Reactive Layer Activation in Context-Oriented Programming

Tetsuo Kamina
Ritsumeikan University
Japan
kamina@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology
Japan
aotani@is.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology
Japan
masuhara@acm.org

Abstract

There are context-dependent behaviors that are active only when a certain condition holds, and that require a certain transition process before activation. We propose a layer-activation mechanism of context-oriented programming languages for such context-dependent behaviors. Our mechanism supports the implicit layer activation (as opposed to the event-based layer activation) in a sense that a condition of activation is written as a conditional expression over *reactive values* (e.g., values obtained from sensors). In addition, it is *push-based* in a sense that it executes the transition process immediately after the condition becomes valid (as opposed to the mechanisms that defer the transition process until the first execution of a context-dependent behavior). In this paper, we present how this mechanism works in an extension of ServalCJ with push-based reactive values, and identify open issues raised by this proposal.

Keywords Implicit layer activation; Transition processes; Reactive values

ACM Reference format:
Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2017. Push-based Reactive Layer Activation in Context-Oriented Programming. In *Proceedings of COP'17, Barcelona, Spain, June 19-20, 2017,* 5 pages.
DOI: 10.1145/3117802.3117805

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 [5, 9]. 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.

There are cases where such context-dependent behaviors are active only when a certain condition holds, and that require a certain transition process that should be executed immediately when the layer is activated and deactivated. For example, we can consider a smartphone application that changes its layout (context-dependent behavior) according to the screen's orientation. In this application, this orientation (context) is represented as a condition over sensor values, and the process that rotates the display and changes the layout (transition process) should be performed immediately when the sensors detect a new orientation.

There have been COP constructs to achieve such requirements. Implicit layer activation has been proposed in the existing COP languages [3, 13, 18] to represent such condition-based layer activation. The transition processes are also supported by existing COP languages [12, 13].

Unfortunately, existing implicit layer activation mechanisms are incompatible with the immediate execution of the transition processes. This is due to the evaluation strategy of the conditional expression in the implicit layer activation. The existing languages adopt the *pull* strategy where the conditional expression is evaluated only when one of the layered methods (i.e., methods that have partial methods) is called. This strategy is based on the assumption that the layer activation affects only method dispatch, which contradicts with the immediate execution of the transition processes. In this strategy, the layer activation is delayed until the first execution of one of the layered methods, which also postpones the execution of the transition processes. Thus, the display is not rotated at the time the sensor detects a new orientation, which is not the behavior that the programmer expects.

To address this problem, we propose a push-based reactive layer activation mechanism where a specified layer is immediately activated/deactivated when the condition changes. To realize the reactive layer activation, we utilize *reactive values* in the conditions for layer activation. The layer activation is considered as an effect of the update of the reactive values. We discuss how this mechanism addresses the aforementioned problem in an extension of ServalCJ [13], which supports both implicit layer activation and transition processes, with reactive values introduced by SignalJ [11]. We also provide some open issues that are raised by this proposal.

The remainder of this paper is structured as follows. Section 2 revisits the background of COP and identifies the problem. Section 3 introduces existing work on which this paper is based, and proposes a push-based reactive layer activation mechanism. This section also shows how the push-based reactive layer activation addresses the aforementioned problem. Section 4 provides some open issues raised by this proposal. Section 5 discusses the related work. Finally, Section 6 concludes.

COP'17, Barcelona, Spain

DOI: 10.1145/3117802.3117805

2 Implicit Layer Activation, Transition Processes, and Their Problems

In this section, we revisit COP mechanisms and describe our motivation.

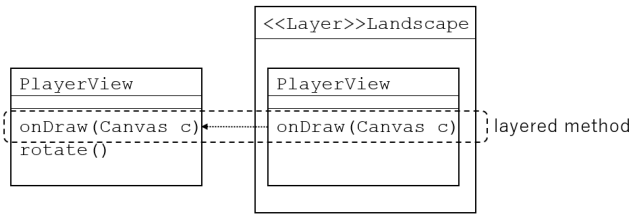

Figure 1. Layers, partial methods, and layered methods. The layer `Landscape` overrides the behavior of `PlayerView` if it is activated. A layered method is a method that includes partial methods. In this figure, `onDraw` is a layered method, while `rotate` is not.

```
contextgroup Orientation(Sensor sensor) {
  activate Landscape if(sensor.value > THRESHOLD);
  activate Portrait when !Landscape;
}
```

Figure 2. Implicit layer activation in ServalCJ

2.1 Layers

Context-oriented programming (COP) provides a modularization mechanism for implementing related context-dependent behavior, which crosscuts several existing classes, into a single module, which is called a *layer*. Furthermore, each layer can be dynamically composed and decomposed with the application, which we call layer *activation* and *deactivation*, respectively. Each layer consists of a set of *partial methods*, which changes the behavior of the original methods in classes when the specified layer is activated. In this paper, we also use an additional terminology to call a method whose behavior is changed by some partial methods as a *layered method* (Figure 1).

For example, we consider a smartphone application that changes its layout according to the screen's orientation. There are two contexts, namely, portrait and landscape, and in this application, we have two corresponding layers, namely, `Portrait` and `Landscape`. As an Android application consists of a set of `Activity`s, and each of them may provide different ways of layout for each portrait and landscape contexts, these context-dependent behaviors are crosscutting concerns and layers modularize them. When the value of the orientation sensor changes to exceed the threshold, the other layer is activated to change the layout of the application.

2.2 Implicit Layer Activation

A number of layer activation mechanisms have been proposed to date: including *per-control-flow* activation, which activates specified layers using `with`-blocks [1, 5], *imperative* activation, which activates layers using imperative commands [7, 8], *event-based* activation, which activates specified layers using declarative events [2, 12], *implicit* activation, which activates specified layers using conditionals [3, 18], and the generalized layer activation mechanism [13].

Among them, the implicit layer activation mechanism provides an intuitive way to specify when the layer is activated using a conditional expression. While other activation mechanisms require that the places where activation takes place in the base program are explicitly specified, the implicit layer activation provides an intuitive mapping from conditions that represent contexts to the activated layers. For example, the activation of `Landscape` is specified as a condition of the orientation sensor.

We show this activation using an example written in implicit layer activation provided by ServalCJ [13] (Figure 2), whose syntax is of the form:

```
activate LAYER if(exp);
```

where `LAYER` is the name of layer and `exp` is an expression of type `boolean`. This code fragment specifies the activation of the layer `Landscape` using the `activate` declaration. The construct `contextgroup` declares a set of related `activate` declarations in that they specify layer activation for some specified group of objects. The condition is specified as an expression followed by the keyword `if`; i.e., `Landscape` is activated whenever `sensor.value > THRESHOLD` returns `true`, and is deactivated otherwise. The activation of `Portrait` is specified using the activation of `Landscape`; i.e., `Portrait` is activated when `Landscape` is not activated.

2.3 Transition Processes

Sometimes, layer activation requires some transition processes after the activation. For example, when the activated layer is switched from `Portrait` to `Landscape`, the current display should be rotated and the layout should be changed *immediately*. To describe this transition process, some COP languages such as EventCJ [12] and ServalCJ [13] provide `activate` blocks[1]. For example, to rotate the display when `Landscape` is activated, we can specify the following `activate` block within that layer:

```
layer Landscape {
  class PlayerView extends View {
    activate {
      rotate(); // Rotate the display and
      invalidate();  // change the layout
    }
  }
}
```

When `Landscape` is activated, this activate block rotates the display and changes the layout.

2.4 Problems in Implicit Layer Activation

One issue in realizing implicit layer activation is determining when the condition is evaluated. Semantically, this condition is evaluated at the every execution step, which imposes a huge amount of overheads on the program execution. Thus, basically, the existing COP languages follow the *pull* strategy: the layer activation is determined when one of the layered methods is called. Thus, when the sensor value changes to exceed a threshold, the layer activation is delayed until one of the layered methods is called (Figure 3). This strategy is based on the assumption that the layer activation affects only method dispatch, and thus this delay does not change the language semantics.

However, this strategy does not work if there is a transition process that should be executed immediately when the activated layer is switched. For example, when the sensor value changes to exceed the threshold, the transition process to rotate the display and change the layout has to be executed, and this transition process

[1] Similarly, transition process that is executed at the time of layer deactivation is written in `deactivate` blocks

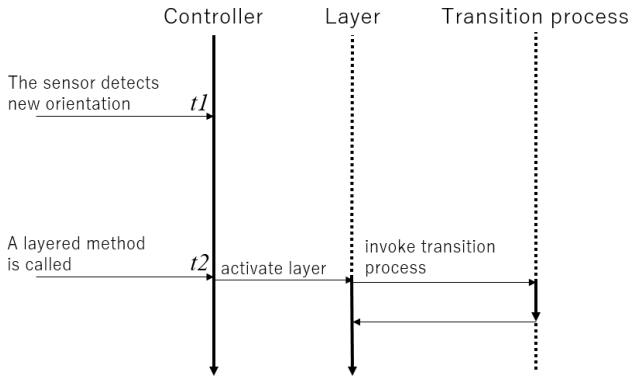

Figure 3. Pull-based implicit layer activation. Rotation of the display and layout change do not occur until $t2$, even though the sensor detects new orientation at $t1$.

is specified in the `activate` block of the related layer. If the layer activation is delayed, the execution of this transition process is also delayed, which is not the behavior that the programmer expects.

Another problem in implicit layer activation is that it makes it difficult to perform a static analysis to determine when the specified layer is activated. For example, determining the control-flow graph nodes where some layers are activated and deactivated is useful for checking required properties between layers, such as exclusions and dependencies between layers [17]. Using implicit layer activation, we need to conservatively analyze every execution points that can change the value of the conditional, which might be imprecise and requiring a whole program analysis.

3 Push-based Reactive Layer Activation

The abovementioned problems call for another evaluation strategy for conditions in the implicit layer activation. We propose a push-based reactive layer activation mechanism where the condition is written using reactive values. We extend ServalCJ [13], a COP language with generalized layer activation, to support reactive values, and explain how this extension addresses the problem.

3.1 Reactive Values

We briefly explain reactive values introduced to ServalCJ. This mechanism is based on SignalJ [11]. In this extension, reactive values are called *signals*, which are used to represent functional dependencies between values that are updated during computation in a declarative way. A signal is stored in a variable declared with the `signal` modifier. For example, the following code fragment declares two signals, `a` and `b`, where signal `b` depends on `a`:

```
signal int a = 5;
signal int b = a + 3;
a++;
System.out.println(b);  // display 9
```

We refer to a signal that depends on other signals as a *composite signal*, which represents a functional dependency between signals. A signal depends on all signals that appear in the right-hand side of the initialization (=) of the signal. For example, `b` in the above code fragment is a composite signal that depends on `a`; i.e., when the value of `a` is updated, this update is propagated to `b`, resulting in the update of the value of `b`. This dependency is fixed during the execution. This means that reassignment of a value to `b` is not allowed in SignalJ, and the value of `b` is updated only through the update of `a`.

On the other hand, `a` in the above code fragment does not depend on any other signals. This is considered a source of the signal network, and its value can be *imperatively* updated at runtime. We refer to this signal as a *source signal*. The change in the source signal is implicitly propagated to other signals that depend on this source signal. Thus, initially the value of `b` in the above code fragment is 8, but after that the value of `a` is updated by `a++`, the value of `b` becomes 9. The update of all the dependent signals is performed at the time of the update of the source signal. Thus, the update of `b` simultaneously occurs when `a` is updated, and 9 is displayed when `println` is called after `a++`.

An update of a signal is considered an event. For example, we can implement an event handler that responds to an update in the signal. An event handler is a lambda expression or a method reference that is passed to the `subscribe` method called on the signal. The following code fragment shows an example:

```
signal int a = 5;
a.subscribe(e -> System.out.println(e));
a++;  // display 6
```

The handler is called whenever the signal is updated. Thus, the lambda expression passed to the `subscribe` is called at the subsequent `a++`, and the value of `a`, which is now 6, is displayed. The formal parameter represents the value of the signal when the handler is called.

We note that, in this data-driven (*push*) [6] computation, the data dependency that is necessary to propagate the updates is determined statically by traversing the declarations of composite signals; i.e., every update of every signal in the right-hand side of the signal declaration is propagated to the declared (left-hand side) signal[2].

The backend of SignalJ is implemented using RxJava2[3]. A source signal is implemented using `BehaviorProcessor`, and a composite signal is implemented using `Flowable`. The event handler (subscriber) is also implemented using the subscription mechanism in RxJava2. Furthermore, the propagation and event handlers can be executed in the different threads using the scheduler mechanism provided by RxJava2.

3.2 Reactive Layer Activation

We show the push-based reactive layer activation (as opposed to the existing ServalCJ strategy) that lifts the conditional expression in `if` to `signal boolean` when that contains at least one signal. We add the following syntax:

```
activate LAYER if(sexp);
```

where `sexp` is an expression of type `signal boolean` that contains at least one signal. We refer to the example of implicit layer activation in Figure 2. First, when signals are used in the conditional

[2] SignalJ also supports the *pull* strategy, i.e., the value of the signal is evaluated at the time the signal is accessed, and thus the signal networks are not limited to the static ones. However, in SignalJ, every event handler needs to be called in the push-based manner and is supported only by static networks.
[3] https://github.com/ReactiveX/RxJava/tree/2.x

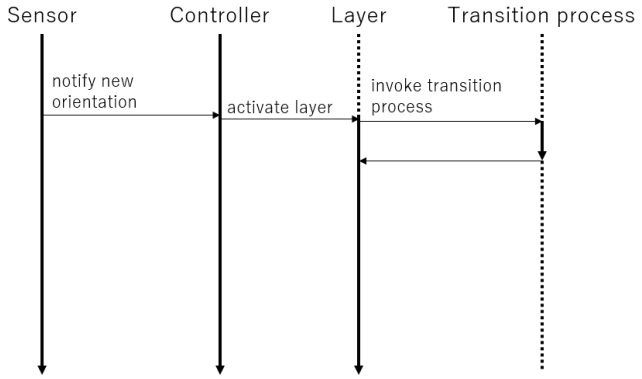

Figure 4. Propagation of updates in the sensor value to the layer activation.

expression in `if`, this conditional expression is considered a composite signal. For example, assuming that the field `value` of `sensor` is declared as a `signal`, i.e., if `value` is declared as follows,

```
class Sensor {
  signal int value = ..; ... }
```

the conditional expression `sensor.value > THRESHOLD` is a composite signal whose value is updated every time the value of `value` changes.

Second, the layer activation is considered an *effect* of the update of the conditional expression. We explain this semantics using a translation from the layer activation in ServalCJ to the event handler of the reactive value. For example, the layer activation in Figure 2 is translated to the following signal and its handler:

```
signal boolean _cond_Landscape =
  sensor.value > THRESHOLD;
_cond_Landscape.subscribe( e ->
  { if (e) activateLayer(Landscape);
    else deactivateLayer(Landscape); } )
```

The signal representing the conditional expression, namely, `_cond_Landscape`, has an event handler that controls the activation of `Landscape`. This handler is called every time the signal is updated. This handler receives the value of the signal at the time of the call as an argument; the handler activates `Landscape` if this value is `true`, and deactivates that layer otherwise.

A subtle issue is when the layer activation is performed, as the handler is executed every time a source signal is assigned a value. The `activateLayer` operation first checks whether the specified layer is activated, and then activates that layer only when it is not activated; otherwise, this operation does nothing. This semantics suppresses unnecessary layer activation, which may change the program behavior because layer activation changes the ordering of activated layers.

We illustrate how this mechanism addresses the aforementioned problem using Figure 4. First, the value of the sensor, which is now declared as a `signal`, changes to exceed the predefined threshold. This changes the value of the condition, which is also a signal, from `false` to `true`. This update calls the event handler, which activates `Landscape`. This activation further executes the transition process described in the `activate` block. This cascade is performed at the time of the update of the sensor value. Thus, the layer activation is not delayed but performed immediately when the sensor detects the different orientation. In other words, the sensor is periodically updated, and every update of the sensor triggers the execution of the event handler, which checks the condition and activates the layer if that condition holds.

We briefly note that where the object `sensor` comes from. A context group can be instantiated using the standard constructor invocation. If the context group declares formal parameters, the constructor invocation has to take arguments for each parameter. In Figure 2, the context group `Orientation` declares one formal parameter, `sensor`. Thus, it is instantiated by providing an instance of `Sensor`:

```
Orientation o = new Orientation(getSensor());
```

We can explicitly specify objects that subscribe this instance (`o`). The layer activation controlled by `o` takes effect on all such subscribers.

We note that, this mechanism also makes it easy to trace the place where the layer activation occurs in the source code, because the dependency between signals is determined statically; i.e., events that may trigger layer activation are determined as updates of statically-known source signals. Thus, our proposal is also compatible with the existing method to detect invalid layer activation using static program analysis [17]. This compensates for the implicit layer activation where determining the layer activation points in the source code is difficult in general.

4 Open Issues

The above discussion shows that the push-based reactive layer activation addresses the problem of implicit layer activation that is used with transition processes. As this work is still in its early stage of the research, there are also a number of open issues, which are described as follows.

4.1 Interoperability with Existing Framework

One issue arises when the proposed language is used with the existing framework that does not utilize the reactive values. For example, the sensor values obtained by Android SDK are not provided as reactive values. Ideally, it is desirable that such time-varying values are provided as reactive values. However, we may also take another approach. Most reactive languages provide *lifting* that converts non-reactive values to reactive ones. Using this mechanism, we can easily implement a wrapper that lifts existing sensor values to reactive values. For example, by periodically updating the reactive values by lifting legacy sensor values in a different thread, we can provide a sensor-monitoring thread with reactive values.

4.2 Interferences between Layer Activation and Reactive Values

Executing a transition process as an effect of an update of a reactive value raises another issue when another update of the reactive value occurs within the transition process. This update within the transition process may trigger another layer activation or deactivation. This can be problematic. For example, if the transition process of an `activate` block triggers deactivation of a layer, and if another transition process of a `deactivate` block of that layer triggers activation of the same layer, this cascade of activation/deactivation results in an infinite loop. As this loop looks unintentional, it should be avoided.

One drastic solution to this problem is to prohibit updates of reactive values within the transition processes. However, updates of reactive values (and cascading layer activation) within the transition processes might be useful in some cases. Thus, simply prohibiting any updates of reactive values within the transition process might be undesirable.

To understand the essence of this issue, we first need to distinguish the intentional updates within the transition process from the unintentional ones. It would be better if we can formally define such unintentional updates. Then, static analyses would be defined on the basis of this formalization.

4.3 Interruptible Execution

Transition processes are related with interruptible execution [3] in that the interruption process may be specified in the transition process in our setting. In the interruptible execution, the context-dependent behavior can be interrupted when the context changes, and can be resumed after that context becomes active again. One issue arises when this resuming process consists of the execution of multiple layered methods, as the existing interruptible execution can resume only the interrupted layered method. The transition process in response to an update of a reactive value can specify such multiple layered methods and thus might help, though the interruption and resuming mechanisms in our setting remain as future work.

4.4 Performance

The final issue is regarding performance. In some cases where updates of reactive values frequently occur, the push-based approach is not advantageous against the pull-based approach with respect to the execution performance. To study this issue, we first need to evaluate the performance overhead imposed by the push-based reactive activation using couples of applications. This evaluation includes microbenchmarking and profiling. If there is some overhead that should not be overlooked, then we may equip some optimization techniques such as pushing updates only when they are necessary. This direction of research also remains as future work.

5 Related Work

Inoue and Igarashi proposed a layer activation mechanism that utilizes reactive values [10]. As in our approach, in this mechanism, the reactive value is used as a condition that determines the layer activation. However, this reactive value is evaluated when one of the layered methods is called. Thus, this mechanism actually takes the pull-strategy, as in the other implicit layer activation, and does not solve the problems identified in this paper. Nevertheless, the same mechanism can also be implemented in the setting of their approach, as their reactive values are taken from signals in REScala [15], which provides the push-based signals and conversions between signals and events. We can define an event that is converted from the signal used in the layer activation specification. Then, we can register an event handler that executes the behavior in the `activate` block to that event.

Our proposal bridges a gap between event-based activation and implicit one in that, while the layer activation is specified as a conditional expression, it is performed in an event-driven manner by considering an update of the conditional expression as an event. A number of effectful reactive values have been proposed in languages that are not Java-based [4, 14, 16]. We believe that those proposals can be adopted to implement our proposal in other host languages, as those provide a mechanism to represent an effect of an update of a reactive value, and our proposal shows that layer activation is realized as an effect of such an update.

6 Concluding Remarks

This paper has proposed a push-based reactive layer activation mechanism, and demonstrated how this mechanism works in the extension of ServalCJ with reactive values. The problem of the implicit layer activation, i.e., the transition process is not executed at the time of an update of the reactive value, is addressed by propagating that update and realizing layer activation as an effect of that propagation. This proposal is considered as a bridge that relates implicit layer activation with the event-based one, which compensate for the disadvantage of the implicit layer activation. This paper also identifies open issues that will develop further research.

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] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Proceedings of the International Conference on Software Composition 2010 (SC'10)*, volume 6144 of *LNCS*, pages 50–65, 2010.

[3] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. Interruptible context-dependent executions: A fresh look at programming context-aware applications. In *Onward! 2012*, pages 67–84, 2012.

[4] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84:108–123, 2015.

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

[6] Conal Elliott. Push-pull functional reactive programming. In *Haskell'09*, pages 25–36, 2009.

[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'10*, volume 6563 of *LNCS*, pages 246–265, 2011.

[8] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object systems. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

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

[10] Hiroaki Inoue and Atsushi Igarashi. A library-based approach to context-dependent computation with reactive values. In *MODULARITY Companion'16*, pages 50–54, 2016.

[11] Tetsuo Kamina. Introducing lightweight reactive values to java. In *SPLASH Companion'16*, pages 27–28, 2016.

[12] 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.

[13] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Generalized layer activation mechanism for context-oriented programming. *LNCS Transactions on Modularity and Composition I*, 9800:123–166, 2016.

[14] Gergely Patai. Efficient and compositional higher-order streams. In *WFLP 2010: Functional and Constraint Logic Programming*, volume 6559 of *LNCS*, pages 137–154, 2010.

[15] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *MODULARITY'14*, pages 25–36, 2014.

[16] Christopher Schuster and Cormac Flanagan. Reactive programming with reactive variables. In *MODULARITY Companion'16*, pages 29–33, 2016.

[17] Noriyuki Suzuki, Tetsuo Kamina, and Katsuhisa Maruyama. Detecting invalid layer combinations using control-flow analysis for android. In *COP'16*, pages 27–32, 2016.

[18] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *ICDL '07: Proceedings of the 2007 International Conference on Dynamic languages*, pages 143–156, 2007.