Generalized Layer Activation Mechanism for Context-Oriented Programming

Tetsuo Kamina[1], Tomoyuki Aotani[2], and Hidehiko Masuhara[2]

[1] Ritsumeikan University, 1-1-1 Noji-higashi, Kusatsu, Shiga, 525-8577, Japan
kamina@acm.org
[2] Tokyo Institute of Technology, 2-12-1 Ohokayama, Meguro, Tokyo, 152-8550, Japan
aotani@is.titech.ac.jp, masuhara@acm.org

Abstract. Context-oriented programming (COP) languages modularize context-dependent behaviors in multiple classes into layers. These languages have *layer activation mechanisms* so that the behaviors in layers take effect on a particular unit of computation during a particular period of time. Existing COP languages have different layer activation mechanisms, and each of them has its own advantages. However, since these mechanisms interfere with each other in terms of extent (time duration) and scope (a set of units of computations) of activation, combining them into a single language is not trivial. We propose a generalized layer activation mechanism based on *contexts* and *subscribers* to implement the different activation mechanisms in existing COP languages in a single language called ServalCJ. We formalize the operational semantics of ServalCJ as a small calculus and prove *priority preservation*, i.e., ensuring that layer prioritization, which resolves the interference between layers, is preserved during computation. To prove this property, we give a formal definition of layer priority that is general so that we can discuss the priorities of layers in other COP calculi and implementations. We implement a ServalCJ compiler, and demonstrate its effectiveness through several example applications.

Keywords: Contexts and subscribers; ServalCJ; Priorities of layers; Priority preservation

1 Introduction

A large number of software systems, such as ubiquitous computing systems, adaptive user interfaces, and self-adaptive systems, as well as their associated computations, require the ability to change behavior with respect to context. For example, for some computations comprising a system, a specific system state that affects such computations may be considered a context. For the system itself, a specific state of the external environment can be considered a context. Dynamic changes in behavior with respect to context changes result in complicated system structures and behaviors that are difficult to predict with traditional programming abstractions.

Context-oriented programming (COP) [17] addresses this difficulty in that it can abstract behavior depending on the same context as a module called a *layer*, and it provides *layer activation mechanisms* so that the behavior in the layer takes effect on a particular unit of computation during a particular period of time. A number of COP languages have been developed to date, and they have successfully modularized such context-dependent behavior [6, 8, 12, 14, 21, 24, 29, 32].

However, existing COP languages have different layer activation mechanisms, making them rather use-case-specific. These layer activation mechanisms have been developed to specify context changes such that they are triggered by internal state changes in the program or external events, or are encoded in the application frameworks. Programmers must select an appropriate mechanism based on use cases. Furthermore, existing layer activation mechanisms are hard-wired into the language and thus do not provide means to extend themselves when combined with other mechanisms in other languages. For example, the per-control-flow activation in ContextJ [6] and JCop [8] is strongly coupled with the current execution thread. Similarly, the implicit activation mechanism in PyContext [32] cannot represent per-instance layer activation. This issue is exacerbated by the fact that different use cases can coexist in the same application. Thus, there is a natural requirement to generalize existing layer activation mechanisms into a single mechanism.

This paper aims to propose a generalized model of layer activation mechanisms that covers all existing COP languages, and to develop a COP language based on that model. To do this, we must solve two problems. First, we must provide a general model to specify a context and the units of computation to which it is applied. Generally, a context can be defined as "everything that exists *outside* the particular unit of computation on which we are focused." However, this definition is too vague when discussing a model on which a particular COP language is based. Second, when developing a generalized COP language, we must unify existing COP mechanisms that may interfere with each other. Thus, we must resolve this interference in order to satisfy programmer expectations.

We tackle these problems by proposing a model based on two concepts: *contexts*, which specify the extent (time duration) of layer activation, and *subscribers*, which specify the scope (a set of units of computations) of activation. These concepts reveal that existing layer activation mechanisms can be explained uniformly using a single model. Furthermore, we define the dynamic semantics of layer activation in the model that satisfy programmer expectations when different existing activation mechanisms coexist in the same application. In the proposed model, the interferences between existing COP mechanisms are resolved by unifying per-instance and global activations, as well as by determining the priority of active layers that are activated synchronously as well as asynchronously.

Based on this model, we have designed the ServalCJ language. A context in ServalCJ is defined as a term of simple temporal logic with a call stack that can represent the extent of layer activation specified by all existing layer acti-

vation mechanisms (to the best of our knowledge). Each context can also be parameterized, which allows us to easily specify the behavioral changes reactively triggered by state changes in the system. A subscriber in ServalCJ is the object on which we focus when considering the context. A set of subscribers can also be global (i.e., all objects are implicitly subscribed to a specific set of contexts when created). A *context group* in ServalCJ specifies a combination of contexts and subscribers.

We demonstrate the effectiveness of ServalCJ through several example applications. The first example is a context-aware program editor, where each construct in ServalCJ is explained. We also present a case study of a maze-solving robot simulator to study the usefulness of ServalCJ. This simulator has different layer activation scenarios, some of which are supported by existing languages, but others are not. We demonstrate that such scenarios are represented uniformly by ServalCJ.

We formalize the dynamic semantics of ServalCJ as a small calculus, Featherweight ServalCJ (FSCJ), to describe how the generalized layer activation is performed. We formulate the *priority preservation* property by stating that the priorities of layers assigned for different layer activation mechanisms are preserved during computation, and prove this property. This formulation is general so that we can discuss the priorities of layers in FSCJ and other COP calculi such as ContextFJ [18, 19] and context holders [4], as well as in other COP implementations with multiple layer activation mechanisms, such as ContextJS [24]. We also show that FSCJ is parameterized over the priority assignment, i.e., we can obtain another calculus that conforms to another priority assignment by changing only some auxiliary definitions and without changing the main part of the reduction rules.

To study ServalCJ's feasibility, we implemented a ServalCJ compiler. The compiler translates ServalCJ programs into standard Java bytecode; thus, they can be run on standard Java virtual machines. We evaluated method dispatch performance in ServalCJ by comparing the time of method calls with and without active layers in ServalCJ against that in plain Java. The results show that our compiler does not impose a serious overhead on the running application.

The remainder of this paper is organized as follows. In Section 2, we introduce an example of a context-aware program editor and review existing COP mechanisms. In Section 3, we argue the necessity of a generalized activation mechanism and explain the challenges in achieving this. In Section 4, we present a model of a unified activation mechanism, and discuss the appropriate dynamic semantics of layer activation. In Section 5, ServalCJ, an instantiation of the model discussed in Section 4, is proposed. In Section 6, we present a case study of a maze-solving robot simulator, compare COP with other implementation techniques, and compare ServalCJ with existing COP languages. In Section 7, we formalize the operational semantics of ServalCJ, provide a definition of layer priority, and prove the priority preservation. In Section 8, we discuss the implementation of the ServalCJ compiler and evaluate its performance. Section 9 discusses related work and Section 10 concludes the paper.

2 Existing COP Mechanisms

In this section, we use an example to explain the commonalities and differences among existing COP languages.

2.1 Example

CJEdit, first implemented by Appeltauer et al. [7], is a program editor that enhances the readability of programs by providing different text formatting techniques for code and comments. The code part is rendered in a typewriter format with syntax highlighting and the comment part is rendered in rich text format that supports multiple fonts, text sizes, decorations, and alignments. Furthermore, CJEdit provides different GUI components depending on which part of the code or comments the programmer is currently editing. For example, when the programmer is editing code, CJEdit displays an outline view of the program so that they can easily determine the structure of the program; when the programmer is editing comments, it displays tools and menus for changing text fonts, sizes, etc.

We extend the CJEdit program editor to make it is multi-tabbed so that the programmer can open multiple files simultaneously. As in the original CJEdit, each tab displays the source code rendered in different text format for code and comments, and different GUI components are provided depending on the cursor's position on the focused tab. A tab displaying an unsaved file shows a mark indicating that the file has not been saved. If the programmer attempts to close a tab that displays an unsaved file, a dialog stating that the programmer is attempting to close an unsaved file is displayed.

We also extend this editor with a couple of features. First, when the editor is used online, the files are stored in a remote repository. When no networks are available, an icon is displayed indicating that the system is operating offline and files are stored on the local disk. Second, we have added a find-name function to CJEdit that can be used to search for the names of variables, methods, and classes throughout the entire source code. During the search, the the mouse cursor changes, and a new widget that displays the status bar is added.

2.2 Overview of COP

In the above example, there are a number of behavioral variations that depend on situations, such as the position of the cursor, rendering of text regions, status of the opened file (saved or unsaved), and the availability of a network. In the following, we refer to such situations as contexts. A COP language provides a modularization mechanism for implementing related context-dependent behavior into a single layer and a layer activation mechanism for dynamically composing and decomposing layers with the application.

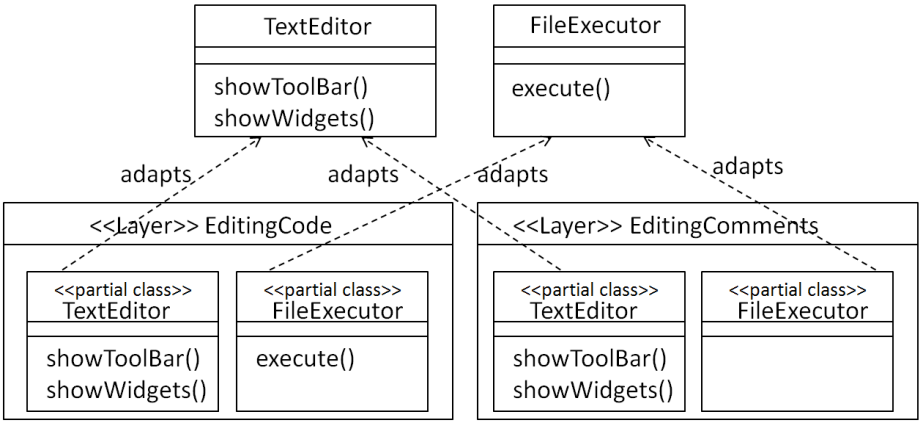


Fig. 1. Relationships among layers and classes

Layers Fig. 1 shows how the related context-dependent behavior is modularized into a layer using a class diagram. The diagram uses two layers, `EditingCode` and `EditingComments`, to represent behavioral variations that are executable only when the cursor is on code or comments, respectively. A COP layer contains a set of partial methods. In Fig. 1, we represent a set of partial methods as a class stereotyped as `<<partial class>>`. A partial method is executable only when the enclosing layer is *active*, i.e., the layer is composed with the application and changes the behavior of the class to which it is applied (Fig. 1). For example, when the `EditingCode` layer is active, at the `TextEditor.showWidgets()` call, the `showWidgets` partial method declared in `EditingCode` is called instead of the original method. In fact, a partial method runs before or after the execution of the original method when it has a `before` or `after` modifier, respectively. If a partial method has no such modifiers, it is called an *around* partial method and runs instead of the original method. Within an around partial method, we can invoke a special `proceed` method to execute the original method. As discussed in Sections 2.3 and 4.2, multiple layers can be active simultaneously, and in that case, when `proceed` is invoked, the partial method in the layer with lower priority is executed.

Layer Activation As mentioned above, a layer can be composed and decomposed dynamically with the running application. These processes are called *layer activation* and *layer deactivation*, respectively. Each COP language provides different linguistic mechanisms to perform activation and deactivation, and this is discussed in the following.

2.3 Different Mechanisms for Layer Activation

Whereas most COP languages provide similar mechanisms for layers, for layer activation, existing COP languages provide a variety of mechanisms. Each mech-

anism differs according to its time period, trigger, and the computations affected by the layer activation.

Per-Control-Flow Activation. One method to activate layers is to use a `with`-block that activates specified layers only within the dynamic scope of the block [6, 8, 12]. For example, we can activate the `EditingComments` layer, which defines behavioral variations that are executable only when the user is editing comments, using the `with`-block.

```
with (EditingComments) { showWidgets(); }
```

The trigger of the layer activation is the computation itself, and its effect continues until the computation leaves the control flow specified by the `with`-block. We note that each `with`-block is implicitly coupled with the currently executing thread and only that thread is affected by the `with`-block.

Another feature of the per-control-flow activation is that, in this model, a programmer is likely aware of the activation order of layers. For example, we can write the following nested `with`-blocks.

```
with(EditingComments) {
  with(RenderingCode) { format(..); }
}
```

This code activates both the `EditingComments` and `RenderingCode` layers, and the inner `with`-block supersedes the outer one. Thus, if these layers define the same partial methods, those defined in `RenderingCode` have priority, i.e., the `before` partial methods in `RenderingCode` are executed first, `after` partial methods in `RenderingCode` are executed last, and around partial methods in `RenderingCode` override those defined in other layers.

Imperative Activation. Some COP languages provide *imperative activation* that uses imperative operations to activate behavior that indefinitely affects the rest of the execution [14, 15]. For example, in Subjective-C [14], the activation and deactivation of a layer is written as follows.

```
[CONTEXT activateContextWithName: @"EditingCode"];
[CONTEXT deactivateContextWithName: @"EditingComments"];
```

The first line activates the `EditingCode` layer, and the second line deactivates the `EditingComments` layer. The activation continues indefinitely, or until another imperative operation that explicitly deactivates the layer is executed. In existing COP languages that support this mechanism, the effect of the activation is *global*, i.e., the entire application is affected by the activation. In general, however, we may consider another variation such that the effect is restricted to within the execution thread.

Event-Based Activation. In this model, the trigger of layer activation is an event, and the activation continues until another event that deactivates the layer is

generated. Unlike activation with an imperative model, this activation can be per-instance and the event receivers may differ from the event senders.

EventCJ [21] supports this model. In EventCJ, an event is declaratively defined using AspectJ-like pointcut language.

```
event MoveOnCode(TextEditor e)
  :after call(void TextEditor.onCsrPosChanged())
    && target(e) && if(e.isCursorOnCode())
  :sendTo(e);
```

This event definition specifies that the `MoveOnCode` event is generated immediately after the `onCsrPosChanged` method call declared in the `TextEditor` class and only if the `isCursorOnCode` call on the receiver object of the former call returns `true`. The `sendTo` clause specifies that this event is sent to only `e`, the receiver of the `onCsrPosChanged` call as specified by the `target` pointcut. In other words, EventCJ supports *per-instance* layer activation. If the `sendTo` clause is omitted, the event is sent to the entire application. Thus, EventCJ also supports global layer activation.

Layer switching upon event is specified declaratively using the layer transition rule.

```
transition MoveOnCode:
  EditingComments ? EditingComments -> EditingCode
  | -> EditingCode;
```

This rule is interpreted as follows. When `MoveOnCode` is generated, if the `EditingComments` layer is active, it is deactivated and `EditingCode` is activated; otherwise, no layers are deactivated and `EditingCode` is activated.

One problem with per-instance activation in EventCJ is that it can only specify instances that are accessible from the join-point where the event is generated. If these instances cannot be obtained from the join-point directly, we must either specify a complex chain of method calls or provide a workaround to access the receiver instances in the base program.

Implicit Activation. In contrast to the above activation mechanisms, where variations of context-dependent behavior are *explicitly* activated, in the *implicit activation* model, the trigger and time period of activation are implicitly specified by a condition. This mechanism is supported by PyContext [32], where the activation is specified by implementing the `active` method, which is implicitly evaluated when the layer activation is tested. We show this in Java-like syntax as follows.

```
class TextEditor {
  .. boolean isCursorOnCode() { .. } ..
  layer EditingCode {
    boolean active() {
      return isCursorOnCode(); } ..
  }
}
```

This code fragment illustrates the `TextEditor` class and `EditingCode` layer in the *layer-in-class* manner [5]. The `EditingCode` layer implements the `active` method that is evaluated whenever, for example, a method that consists of a set of partial methods is called, and, if `active` returns `true` (i.e., if the `isCursorOnCode` call returns `true`), the `EditingCode` layer becomes active.

In PyContext, only the currently executing thread is affected by the implicit activation, as in per-control-flow activation.

3 Problem Statements

In this section, we present the expressibility problem in existing COP mechanisms and the interference problem that exists between the mechanisms.

3.1 Expressibility Problem

When we choose one COP language to implement context-dependent behavior, we sometimes encounter difficulties because each mechanism fits only specific cases of behavioral changes in the application. For example, in the CJEdit example, if we choose the per-control-flow model, it becomes difficult to implement event-driven behavioral changes triggered by, for example, a change in the position of the cursor. On the other hand, if we choose the event-based model, it is difficult to implement the find-name function, which recursively searches the name in the entire source code, because the state transition model of the event-based activation cannot represent the call stack. Furthermore, the set of entities affected by the layer activation also varies within the application. For example, the arrangement of widgets and tools in the toolbar and the behavior depending on the network availability are applied to the entire application, while the status of opened files can vary for each tab.

We face similar problems in other context-aware applications. For example, in a multi-tabbed Twitter client, each tab displays the user's timeline, which is updated after a followed person posts a tweet. Each tab behaves differently with respect to contexts, such as tab focus (focused or unfocused) and the content displayed on the timeline (all tweets from all followed accounts, tweets only from a specific account, or all tweets that match a search keyword). The trigger of a context change can be an event, such as clicking a tab, and can be defined implicitly relative to timeline content. The effect of behavior changes may also vary. Each tab can change its behavior dynamically, and its effect is restricted to only the instances contained within the tab. We can also consider other cases, such as behavior changes with respect to battery status, which can affect the entire application. Another example is a pedestrian navigation system that changes behavior with respect to changes in situation, such as moving from an indoor to an outdoor environment, which is triggered by an event. In addition, such a system can change behavior based on changes in computation, such as "during map download," which is activated only within the control-flow.

We also argue that some COP mechanisms provide incomplete abstractions. For example, EventCJ supports per-instance activation, where we can specify only instances accessible from the join-point where the event was generated. Similarly, events in event-based activation in EventCJ are only join-points, and thus EventCJ does not provide any way to abstract the event sender.

3.2 Interference Problem

Some COP languages support multiple activation mechanisms and thus support some combination of different behavioral change use cases in the application. For example, EventCJ supports global activation as well as per-instance activation so that the effect of the behavioral change is exerted on the entire application. Similarly, ContextJS [24] supports global activation as well as per-control-flow activation as pre-defined activation mechanisms. Although these languages allow us to represent different cases of behavioral changes uniformly to some extent, the activations that they support are still limited. For example, neither language supports implicit activations.

A more serious problem with existing approaches is that an activation mechanism sometimes interferes with an activation triggered by another mechanism. There are two interference problems, i.e., between global and per-instance activations and between synchronous and asynchronous activations.

Global-Per-Instance Interference. We explain the former interference problem using an example of a mobile application written in EventCJ that uses both global and per-instance activation mechanisms. Suppose that the layer `BatteryLow`, which implements the "energy-saving mode" behavior that uses less precise computation and fewer resources, is globally active because the battery power of the executing machine is low. Also suppose that activation on some instances is controlled in a per-instance manner to allow the user to control the behavioral changes of these instances manually. For example, the user may require some objects to produce precise computation results in short periods even when the battery is on the verge of running out.

In fact, EventCJ does not support such a situation because global activation always cancels a per-instance deactivation. In EventCJ, the layers activated by global activation and those activated by per-instance activation are stored in different arrays, and the partial method dispatch uses both arrays. Thus, the layer stored in the global activation array is effective even when it is removed from the per-instance activation array. A similar problem also occurs in ContextJS. Although this may be an implementation issue, this kind of interference is likely to arise if the different linguistic mechanisms were "piled up" into a single language.

Synchronous-Asynchronous Interference. Another type of interference occurs when we unify activation mechanisms from different languages. In the per-control-flow model, the order of active layers is explicit for the programmer, i.e.,

the inner-most layer always precedes other layers. Although in other models, such an order is not explicit for the programmer, the order of active layers is also well-defined to make the execution result universal. For example, in EventCJ, the most recently activated layer always precedes the others [3]. This semantics of EventCJ conflict with those of the per-control-flow model in ContextJ. For example, in the following `with`-block, the programmer expects the text block stored in `textBlock` to be formatted with syntax highlighting.

```
SyntaxHighlighter sh = ..
with(EditingCode) {
  with(RenderingCode) {
    // forcing text to be formatted with
    // syntax highlighting
    sh.format(text);  } }
```

However, the event-based layer activation may not meet this expectation because an event activating `EditingComments` may be generated after the activation of `EditingCode` and before the call of `format`, thereby causing the syntax highlighting to be switched off.

The source of this conflict is the mixing of the synchronous layer activation, where the trigger is the computation itself, and the asynchronous layer activation, where the trigger is the external event. If the layer activation is synchronous with the execution of the application described in the base program, the programmer is aware of the execution point when the specified layer becomes active. On the other hand, we cannot foresee when layer activation will be triggered asynchronously by events.

4 Model of Generalized Layer Activation

To address the aforementioned problems, we propose a generalized model of the existing COP mechanisms and provide the semantics of layer activation to define activation order uniformly.

4.1 Contexts and Subscribers

To develop the generalized activation model, we coordinate the different layer activation mechanisms in existing COP languages using the following concepts, i.e., *context*, which specifies the time and duration of the layer activation[3], and *subscribers*, which specifies which computations the activation affects. A number of layer activations represented by contexts affect a specific set of subscribers. We combine a set of contexts with a set of subscribers and call this combination a *context group*. When an object subscribes to a context group, method dispatch on the object includes the partial methods in the active layers with respect to the context group. In other words, when we activate a layer with respect to

[3] We use the term "context" to indicate the temporal context.

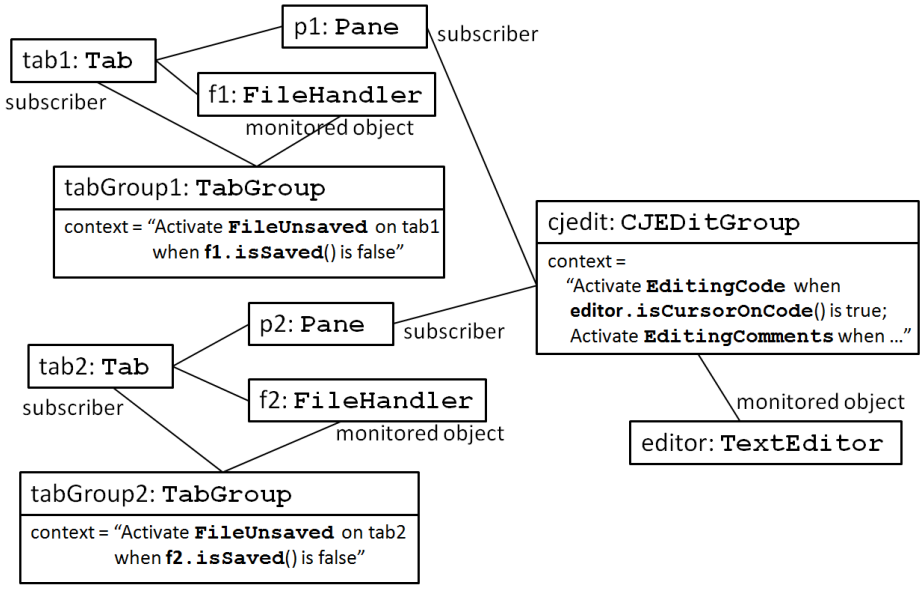


Fig. 2. Unified model of COP in an object diagram. We abbreviate insignificant edges, i.e., every object is a subscriber of the instance `cjedit`. This is not represented in the diagram because only specific instances (`p1` and `p2`) provide context-dependent behavior for `cjedit`.

a context group, all the objects subscribing to that group will begin searching partial methods in that layer upon method dispatch. For example, the contexts that specify when the cursor is on code or comments affect the entire application with respect to the behavior of the toolbar and menubar; thus, they are grouped into a single context group. The context specifying when the opened file in a tab is unsaved affects only a limited subset of instances in the application; thus, they are grouped into another context group.

We illustrate this model in Fig. 2 using a UML instance diagram. In this diagram, the instance `cjedit` of the context group `CJEditGroup` specifies contexts for activating `EditingCode`, which implements the code-editing functions, and `EditingComments`, which implements the comment-editing functions. All instances in the entire application subscribe to this context group. These contexts are parameterized over the objects on which the layer activation depends, e.g., in `cjedit`, this parameter is bound to `editor`, an instance of `TextEditor`. When the state of `editor` changes, the layer activation of all subscribed instances also changes. Similarly, the instance `tabGroup1` of the `TabGroup` context group specifies the contexts for activating `FileUnsaved`, which implements the behavior related to unsaved files. Only the instance `tab1` of `Tab` subscribes to that context group. The context specified in `TabGroup` is also parameterized, and this parameter is bound to `f1`, an instance of `FileHandler`.

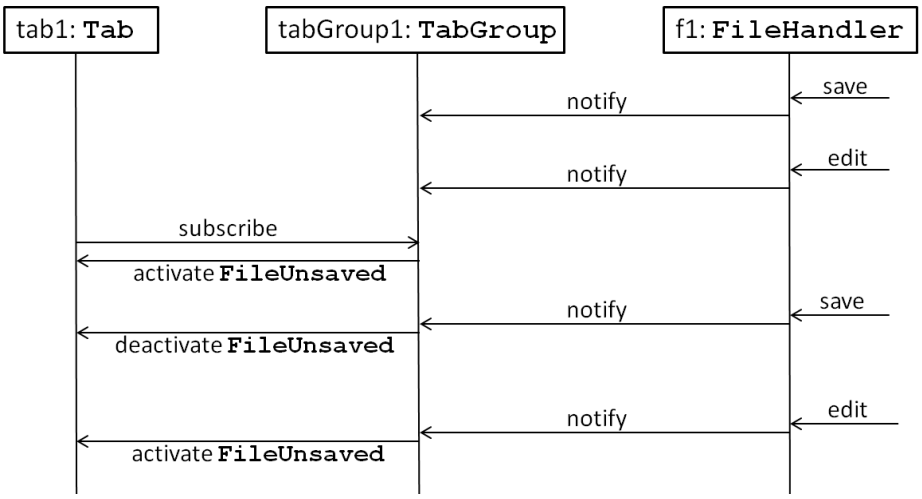


Fig. 3. Dynamic subscription and layer activation

Table 1. Existing COP languages categorized in our model

	Global	**Thread**	**Instance**
Control-flow		ContextJ[6], PyContext[32]	ContextErlang[29]
Imperative	Subjective-C[14]		ContextErlang
Event-based	EventCJ[21]		EventCJ
Implicit	Flute[10]	PyContext	

We further illustrate the dynamic semantics of this model using the UML sequence diagram in Fig. 3. When the instance `f1` of `FileHandler` changes its state according to outside operations such as the "save" and "edit" commands, it also notifies these changes to the instance `tabGroup1` of the context group `TabGroup`, which refers to `f1`. If no instances subscribe to `tabGroup1`, these notifications do not trigger any layer activation. After an instance of `Tab`, i.e., `tab1`, subscribes to `tabGroup1`, it immediately activates `FileUnsaved` on `tab1` if `f1` is not saved after editing. After this subscription, the notifications from `f1` triggered by the state changes on `f1` trigger the activation and deactivation of `FileUnsaved` on `tab1`.

We show that each existing COP language falls into one specific case of this model, as illustrated in Table 1. In Table 1 the methods that specify contexts are categorized into four variants, i.e., per-control-flow, imperative, event-based, and implicit, that correspond to each layer activation model discussed in Section 2.3. In the table, the methods that specify subscribers are also categorized into three variants, i.e., global (the "world"), thread (the currently executing thread), and instance (a limited set of instances). Each cell represents the COP languages that

support the specific combination of these methods. In addition to the languages discussed in the previous section, we also list the COP languages mentioned in Section 10. For example, EventCJ supports event-based specification of contexts that are applicable to both all instances in the application and a limited set of instances. Some cells indicate that no existing COP languages support such a combination. For example, implicit activation of a limited set of instances is not supported by any existing COP language.

4.2 Model of Activation Order

The synchronous-asynchronous interference explained in Section 3.2 implies that we must manage synchronous and asynchronous layer activation separately. To satisfy programmer expectations, in our model, synchronous layer activation always precedes asynchronous activation. More precisely, the semantics of layer activation in our model are defined as follows.

First, we define synchronous and asynchronous layer activation.

- Layer activation is *synchronous* if and only if its context is specified as a control-flow and it is statically known that its subscribers contain the thread that will execute the control-flow. For example, global and per-thread activation with the per-control-flow model are considered synchronous.
- Layer activation that is not synchronous is *asynchronous*.

We then define the order of active layers as follows.[4] Let $\bar{\mathtt{L}}_S = \mathtt{L}_1, \cdots, \mathtt{L}_n$ be a sequence of layers that are activated synchronously, and let $\bar{\mathtt{L}}_A = \mathtt{L}'_1, \cdots, \mathtt{L}'_n$ be a sequence of layers that are activated asynchronously. We assume that there are no duplicate layers in a sequence of activated layers. We define the function *actSync* that takes a concatenation of sequences of activated layers $\bar{\mathtt{L}_A};\bar{\mathtt{L}_S}$ and a layer $\mathtt{L}$ and returns a new concatenation of the sequences of activated layers.

$$actSync(\bar{\mathtt{L}}_A;\bar{\mathtt{L}}_S,\mathtt{L}) = (\bar{\mathtt{L}}_A \setminus \mathtt{L});(\bar{\mathtt{L}}_S \setminus \mathtt{L})\mathtt{L}$$

This function models synchronous layer activation. If $\mathtt{L}$ is not contained in both $\bar{\mathtt{L}}_A$ and $\bar{\mathtt{L}}_S$, it is added at the head of sequence $\bar{\mathtt{L}}_S$, indicating that $\mathtt{L}$ has the highest priority. Otherwise, $\mathtt{L}$ is removed from the original position and is moved to the head of the sequence $\bar{\mathtt{L}}_S$.

Similarly, asynchronous layer activation is modeled by the *actAsync* function.

$$actAsync(\bar{\mathtt{L}}_A;\bar{\mathtt{L}}_S,\mathtt{L}) = \begin{cases} (\bar{\mathtt{L}}_A \setminus \mathtt{L})\mathtt{L};\bar{\mathtt{L}}_S & \text{if } \mathtt{L} \notin \bar{\mathtt{L}}_S \\ \bar{\mathtt{L}}_A;\bar{\mathtt{L}}_S & \text{if } \mathtt{L} \in \bar{\mathtt{L}}_S \end{cases}$$

If $\mathtt{L}$ is not contained in both $\bar{\mathtt{L}}_A$ and $\bar{\mathtt{L}}_S$, it is added at the head of the sequence $\bar{\mathtt{L}}_A$, indicating that $\mathtt{L}$ has higher priority than all layers in $\bar{\mathtt{L}}_A$ but has lower

[4] As illustrated in Section 3.2, we believe that this ordering is preferable in many cases. However, we also acknowledge that it is preferable for programmers to configure the ordering policy in particular cases. This configuration mechanism is discussed in Section 7.

priority than all layers in $\bar{\mathtt{L}}_S$. If L is contained in $\bar{\mathtt{L}}_A$, it is moved to the head of $\bar{\mathtt{L}}_A$. If L is contained in $\bar{\mathtt{L}}_S$, the order of the active layers does not change, because this case indicates that L has already been activated with higher priority than the layers in $\bar{\mathtt{L}}_A$.

We define the function *deact* to model layer deactivation.

$$deact(\bar{\mathtt{L}}_A; \bar{\mathtt{L}}_S, \mathtt{L}) = (\bar{\mathtt{L}}_A \setminus \mathtt{L}); (\bar{\mathtt{L}}_S \setminus \mathtt{L})$$

The above functions are used when we describe the operational semantics shown in Section 7. For example, *actSync* is always used when the `with`-block is applied, and *actAsync* is always used when event-based activation is applied. The order of active layers $\bar{\mathtt{L}}_A; \bar{\mathtt{L}}_S$ is used when dispatching a partial method. The search for a partial method begins from the right-most layer of $\bar{\mathtt{L}}_S$ and proceeds to the left-most layer of $\bar{\mathtt{L}}_A$. If no partial methods are found, the original method is dispatched.

To address global-per-instance interference, every activation is performed in a per-instance manner. This means that, when a layer becomes globally active, that layer is added to the active layers for all instances that have that layer. This mechanism ensures that global activation does not interfere with per-instance activation at the cost of activating the layer for all of these instances.

5 COP Language with Contexts and Subscribers

We have designed the COP language ServalCJ to be an instance of the generalized activation model discussed in Section 4. ServalCJ provides the following linguistic constructs: *activate declaration*, which specifies when the layer is active in terms of *contexts* that identify the extent of layer activation, and *context group declaration*, which modularizes these declarations and specifies the set of *subscribers* where they are applied. In ServalCJ, a subscriber is the object on which we focus when considering the context.

ServalCJ is a layer-based COP language that provides a modularization mechanism for context-dependent behavior using layers. ServalCJ supports the class-in-layer syntax of layer declarations as well as the layer-in-class syntax [5], where we can define a set of partial methods and `activate`/`deactivate` blocks. This paper focuses on how layer activation is specified by ServalCJ; how layers are declared in ServalCJ is beyond the scope of this paper.

We formalize the dynamic semantics of ServalCJ in Section 7. While the formal model provides semantics based on primitive linguistic constructs, ServalCJ provides a more convenient syntax.

5.1 Context Group Declarations

In ServalCJ, a context group is declared using a *context group declaration*. A context group groups related specifications of layer activation into one module, and can be instantiated. Each context group instance contains subscribers, i.e., a

```
1 contextgroup EachTabGroup(FileHandler f) {
2   subscriberTypes: Pane, FileHandler;
3   activate FileUnsaved if(!f.isSaved());
4 }
```

Fig. 4. Context group declaration for CJEdit specifying the layer activation for each tab

set of instances where the specified layer activation is applied. A context group can also declare parameters that can be referred to from the layer activation specification.

Fig. 4 shows an example of layer activation for CJEdit that specifies the layer activation for each tab. Line 1 specifies the name of the context group and its parameter. We can replace this parameter with an argument when this context group is instantiated. A context group is instantiated using the standard `new` expression. We can also declaratively specify when the instance of context group is created using the AspectJ pointcut and advice mechanism. For simplicity, we do not use this mechanism in this paper.

```
FileHandler file = new FileHandler(..);
EachTabGroup etg = new EachTabGroup(file);
etg.subscribe(file);
Pane pane = new Pane();
etg.subscribe(pane);
```

An object can dynamically subscribe to the instance of a context group, thereby becoming one of the subscribers of that context group. This subscription is performed by calling the `subscribe` method on the instance of context group. For example, in the above code fragment, instances of `FileHandler` and `Pane` subscribe to `etg`, which is an instance of `EachTabGroup`. The current version of ServalCJ requires that each context group declares the types of instances that can subscribe to it (Line 2, Fig. 4). We can also declaratively specify which instance subscribes to this context group when using the AspectJ pointcut and advice mechanism. This flexible subscription mechanism addresses the problem of per-instance activation in EventCJ, where any receivers of an event must be accessible from the specified join-point.

Line 3 of Fig. 4 declares when the layer `FileUnsaved` is active, which occurs whenever the `isSaved` method call on `f` returns `false`. We further discuss the specification of layer activation in Section 5.2.

Global Context Groups. In the aforementioned example, we explicitly specified which instances subscribe to the context group. In ServalCJ, we can also declare a context group that affects all instances in the application. Such a group is called a *global context group*.

Fig. 5 shows an example of a global context group declaration. To make the context group global, we must provide the `global` modifier. A global context

```
1 global contextgroup CJEditGroup(TextEditor e) {
2   activate EditingCode if(e.isCursorOnCode());
3   activate EditingComments if(e.isCursorOnComments());
4 }
```

Fig. 5. Example of a global context group

group does not contain any specifications for subscribers. Instead, every object is implicitly considered to have subscribed to the global context group. As for other context groups, we can create an instance of the global context group, which becomes effective only after instance creation. The context group `CJEditGroup` in Fig. 5 declares two layer activation rules: (1) the layer `EditingCode` is active whenever the `isCursorOnCode` method call on `editor` returns `true` and (2) the layer `EditingComments` is active whenever the `isCursorOnComments` method call on `editor` returns `true`.

5.2 Declaring Layer Activation

In ServalCJ, we define when the layer is active by specifying the name of the layer and a Boolean term, i.e., when the Boolean term is `true`, the layer is active. This specification is performed using an *activate declaration*, which has the following syntax.

`activate` *LayerName Context* `;`

This declaration begins with the keyword `activate` followed by the name of the layer. Next, we specify a context, which has the `boolean` type in Java.

In particular, in ServalCJ, a context is declared using a temporal logic term with call stacks. This term consists of *if expressions* that specify the condition under which the context is active, *from-to expressions* that specify the from-event and to-event that activate and deactivate the context, respectively, *cflow expressions* that specify the control flows where that context is active, *named contexts* that are contexts identified by name, and *composite contexts* that are contexts combined by using logical-OR, logical-AND, and NOT expressions. We discuss each of these terms in the following.

Conditional Expressions. The first way to specify layer activation is to use a conditional (`if`) expression that corresponds to implicit activation (Section 2.3). To support implicit activation, ServalCJ provides `if` expressions that specify the condition under which the context is active. We have provided an example in Fig. 4, which contains the following activate declaration.

```
activate FileUnsaved if(!f.isSaved());
```

Within `if` expressions, we can use any Boolean-type Java expression. Note that ServalCJ can represent implicit activation that is applied per-instance. As shown

```
class TextEditor {
  event MoveOnCode;
  event MoveOnComments;
  void onCursorPositionChanged() {
    if (isCursorOnCode()) { MoveOnCode(); }
    else if(isCursorOnComments()) { MoveOnComments(); }
  }
}
```

Fig. 6. Publishing events in ServalCJ

in Fig. 4, we can create a different instance of `EachTabGroup` for each tab that contains distinct instances of `Pane` and `FileHandler`. Each instance of `EachTabGroup` refers to a distinct instance of `FileHandler` through the variable `f`, which is referenced from the `if` expression. Thus, we can control the activation of layers for each tab independently.

From-to Expressions. A from-to expression specifies the *events* that activate and deactivate the context. This expression makes it possible to represent event-based layer activation. An event in ServalCJ is declared as a member of a class and triggered like a method invocation. For example, in Fig. 6, two events, `MoveOnCode` and `MoveOnComments`, are declared in the class `TextEditor`. These events are triggered during the execution of `onCursorPositionChanged` and if the `isCursorOnCode` (`isCursorOnComments`, resp.) call returns `true`. We can also declare an event using the AspectJ pointcut language.

Using these events, we can specify when the `EditingCode` layer becomes active and inactive as follows.

```
activate EditingCode
  from MoveOnCode to MoveOnComments;
```

This declaration specifies a *from-event* that activates `EditingCode` and a *to-event* that deactivates the layer. Here, the `EditingCode` layer is activated whenever the `MoveOnCode` event is triggered and is deactivated whenever the `MoveOnComments` event is triggered.

As in the case of implicit activation, we can specify the *sender* of the event by referring to the parameter of the enclosing context group.

```
contextgroup CJEditGroup(TextEditor editor) {
  activate EditingCode
    from editor.MoveOnCode
    to editor.MoveOnComments;
}
```

This activate declaration specifies that `EditingCode` is activated when `MoveOnCode` is triggered and is deactivated when `MoveOnComments` is triggered *only when these events are triggered by* `editor`. Note that we cannot specify an event sender in EventCJ.

Cflow Expressions. A cflow expression specifies a control-flow in which the layer is active. This expression makes it possible to represent per-control-flow layer activation. An example of a cflow expression is as follows.

```
activate SearchingName
  in cflow(call(void FileHandler.find(*)));
```

This context declaration specifies that the `SearchingName` layer is active only under the control flow specified by the `cflow` expression, which is the entire execution of the `find` method declared in the `FileHandler` class. Note that cflow expressions are not a particular case of from-to expressions, because we cannot represent a control-flow using a from-to expression when the control-flow under the specified method call contains the same method call specified in the cflow expression.

Per-Thread Activation. The `with`-block-based COP languages, such as ContextJ, activate layers in a per-thread manner. Note that most useful cases of ContextJ are easily encoded by a combination of a global context group and cflow activation. To restrict the effects of layer activation to the currently executing thread, we may introduce another modifier, `perthread`, that limits the set of subscribers to the subscribers accessed from the thread executing the control flow.

```
global contextgroup AContextGroup(..) {
  perthread activate ALayer in cflow(..);
}
```

The `perthread` modifier does not have any effect when it is used with other expressions.

Named Contexts. The same contexts are sometimes used in different activate declarations. To improve the reusability of contexts, ServalCJ provides a *named context*, which is a mechanism that provides a name to a context to make it possible to reference it from several activate declarations. A named context in ServalCJ is declared using the following syntax.

`context` *ContextName* `is` *Context* `;`

This declaration begins with the keyword `context` followed by the name and specification of the context. The syntax of the context is the same as that specified in activate declarations. The name of the context is used in activate declarations and should be enclosed within a `when` clause. For example, the context group declaration:

```
contextgroup Highlighter(SyntaxHighlighter sh) {
  context RenderCode is
    if(sh.getBlock().isCodeBlock());
  activate Highlighting when RenderCode;
}
```

is identical to the following declaration.

```
contextgroup Highlighter(SyntaxHighlighter sh) {
  activate Highlighting
    if(sh.getBlock().isCodeBlock());
}
```

ServalCJ also provides a way to compose contexts to represent more complex layer activation. This composition was originally known as *composite layers* [22]. To compose contexts, we can use the logical operators `||` (logical-OR), `&&` (logical-AND), and `!` (NOT).

6 Case Study

The program editor example described above shows how different COP mechanisms coexist in the same application, thereby justifying the design of the generalized layer activation mechanism in ServalCJ.

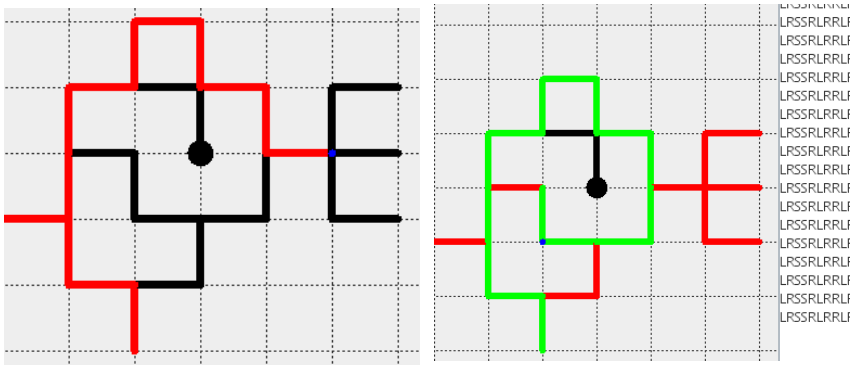
To provide more evidence, we conducted another case study to implement a maze-solving simulator.[5] This application simulates how a line-tracing robot solves a maze. The following code skeleton illustrates how the robot solves a maze.[6]

```
void run() {
  while (!isGoal()) {
    followSegment();
    printPath();
    turn();
    simplify();
  }
}
```

The `followSegment` method performs line-tracing until the robot reaches an intersection, a corner, or a dead-end (in the following, we refer to these as *intersections*). The robot detects an intersection using sensors. The `printPath` method prints some debugging information on the LCD attached to the robot. The `turn` method selects one path from the outgoing paths at an intersection by applying a specific rule (e.g., the left-hand rule selects the left-most path) and controls the motors to make the robot turn accordingly. The `simplify` method calculates a potentially optimized path from the start point to the current intersection by eliminating dead-ends. The robot repeats these behaviors until it reaches the goal. After solving the maze, the robot can run the optimized path

[5] The simulator source code is available at `https://github.com/ServalCJ/mazesimulator.git`.

[6] This case study was inspired by the real maze-solving Pololu 3pi Robot (`http://www.pololu.com/product/975`). The simulator's behavior follows the sample program provided by the 3pi Robot distribution.



(a) Simulator solving a maze (red lines represent the traced path)

(b) Simulator in debug mode (green lines overwrite the red lines and represent the optimized path; the textbox displays the text-based notation of the optimized path)

Fig. 7. The maze-solving simulator (the lines indicate paths within the maze; the black circle represents the goal)

from the start point to the goal by simply following the path calculated by `simplify`.

If the maze contains loops, the robot must remember all visited intersections and/or segments (i.e., a path from one intersection to one of the neighbors) to detect such loops. There are several algorithms to solve mazes; some can only solve mazes that contain no loops, and others can solve mazes with loops.

The simulator emulates the behavior of a maze-solving robot. In this simulator, the maze is modeled as a graph where each node representing an intersection provides coordinates to indicate its position. The instance `robot` of `Robot` emulates maze-solving in this model, e.g., the `followSegment` method simply updates the current position of the `robot` according to the destination of the edge that models the segment. The simulator provides three algorithms to solve the maze, i.e., the left-hand rule, right-hand rule, and Trémaux's algorithm.[7] The selection of these algorithms changes the behavior of `turn` and possibly that of `simplify`.

For the user, this simulator provides a number of functions, i.e., edits a maze, simulates how the robot solves the maze, and simulates how the robot follows the optimized path after solving the maze. These functions are exclusive, i.e., when we are editing a maze, we cannot run any simulations for solving the maze or following the optimized path. These functions are switched when the user finishes editing the maze (or loads a pre-edited maze) and when the robot finishes solving the maze. The simulator provides GUI tools, such as a menubar and menu buttons, that are switched automatically when the functions are switched. During maze-solving, visited intersections and segments are colored to visualize

[7] Among them, only the last algorithm can solve mazes with loops.

the traced path (Fig. 7(a)). Furthermore, while the robot is solving the maze, the user can select a debug mode to display the currently calculated optimized path by printing text that represents the optimized path and changing the color of intersections and segments in the optimized path (Fig. 7(b)).

We implemented this simulator using ServalCJ, and a number of layers were defined to implement context-dependent behavior:

- `EditingMaze` provides GUI tools for editing the maze, such as inserting segments and intersections, saving the maze to a file, opening a maze from a file, and finishing editing the maze.
- `SolvingMaze` provides GUI tools for starting the simulation, solving the maze, stopping the simulation, switching to debug mode, and selecting the algorithm to solve the maze (the default is the left-hand rule).
- `RunningMaze` provides GUI tools for starting the simulation, following the optimized path and stopping the simulation.
- `RightHandRule` solves the maze using the right-hand rule.
- `Tremaux` solves the maze using Trémaux's algorithm.
- `Debugging` provides a textbox to display the currently calculated optimized path.
- `UnderDebugging` changes the color of segments and intersections in the maze only if they are in the optimized path and debug mode is selected.

All of these layers crosscut multiple classes. Even `RightHandRule` and `Tremaux`, which seem to be related to only a single instance of a robot, affect both a robot instance and the GUI tools. Note that the debugging feature is divided into two layers, `Debugging` and `UnderDebugging`, because they are applied in slightly different situations, as will be explained below.

These layers change the behavior of multiple classes. For example, `SolvingMaze` and `RunningMaze` change the appearance of the GUI components and the behavior of the simulator. The simulator is executed in a different thread from the GUI components, and the behavior of the `run` method is switched when the active layer is changed (Fig. 8).

To specify layer activation, we implemented two context groups. The first context group manages layer activations that are applied globally, and the other context group manages layer activations that are applied only to specific instances.

Fig. 9 shows the context group for managing globally activated layers. It specifies activate declarations for five layers. The activation of the first four layers is controlled by from-to expressions. The events that activate and deactivate the layers correspond to the GUI events generated by the operations taken by the user. The `UnderDebugging` layer is a composite layer; it is active only when the `Debugging` layer is active and the additional condition specified by the named context `Print` holds. As Fig. 10 shows, the `UnderDebugging` layer changes how the color of visited segments and intersections is set. First, this behavior is applicable only when the application is in the debug mode. Second, this behavior is applicable only to the intersections and segments in the shortest path. Thus, `UnderDebugging` is activated only in the control flow where the shortest path

```
layer SolvingMaze {
  class Robot {
    public void run() {
      /* maze solving behavior */
    }
  }
  class View {
    public void setMenuBar() { .. }
    public void setButtons() { .. }
  }
}
layer RunningMaze {
  class Robot {
    public void run() {
      /* running the optimized path */
    }
  }
  class View {
    public void setMenuBar() { .. }
    public void setButtons() { .. }
  }
}
```

Fig. 8. Example layers in the maze-solving simulator

```
1 global contextgroup MazeUI() {
2   activate EditingMaze
3     from startEditor to startSolver;
4   activate SolvingMaze from startSolver to solved;
5   activate RunningMaze from solved to neverMatchingEvent;
6   activate Debugging from startDebug to endDebug;
7   context Print is in cflow(call(void Simulator.print()));
8   activate UnderDebugging when Debugging && when Print;
9 }
```

Fig. 9. Context group for globally activating layers

is printed (which also calls the `setTraced` methods on `Edge` and `Node`). In this case, we apply the `cflow` expression.

Fig. 11 shows the context group for managing activations that are applicable to a specific robot instance. Although there is only one robot instance in this application, we apply per-instance activation in this case for future extensibility (e.g., supporting multiple robots that execute different algorithms). In this case, we apply conditional (`if`) expressions rather than from-to expressions to specify the activate declarations because, in the base program, the value indicating the algorithm is set to the robot instance when the user selects the algorithm,

```
layer UnderDebugging {
  class Edge { // segments
    public void setTraced() {
      proceed();
      color = Color.GREEN; //the default is RED
      src.setTraced();
      dst.setTraced();
    }
  }
  class Node { // intersections
    public void setTraced() {
      proceed();
      color = Color.GREEN;
    }
  }
}
```

Fig. 10. The `UnderDebugging` layer

```
1 contextgroup Algorithm(Robot robot) {
2   activate RightHandRule if(robot.isRightHandRule());
3   activate Tremaux if(robot.isTremaux());
4 }
```

Fig. 11. Context group applicable to the robot instance

which is useful for determining which layer should be activated. Note that the program structure of the base program may affect how the programmer selects the activation mechanism.

Discussion. We first discuss the appropriateness of applying COP to implement this simulator.[8] First, the variations of context-dependent behavior in this simulator crosscut multiple classes and are modularized by corresponding layers in COP. For example, the `SolvingMaze` and `RunningMaze` layers change the behavior in both the simulator and the GUI components. `Debugging` changes the appearance of the GUI (showing or hiding the textbox that prints the shortest path) and the behavior of the simulator (whether the shortest path stored in the simulator instance is printed). `UnderDebugging` changes the color of intersections and segments. The algorithms applied to the simulator instance also change the appearance of the GUI components (e.g., the currently selected algorithm is disabled for selection in the menu). Thus, it is appropriate to use layers to implement these behavioral variations.

Second, COP supports disciplined changes of context-dependent behavior. We can apply meta-programming techniques to implement dynamic changes

[8] The same discussion is also applicable to the program editor example.

of behavioral variations. However, in such techniques, it is difficult to mechanize reasoning about some properties among these variations. For example, in this simulator, the variations of behavior implemented in `EditingMaze`, `SolvingMaze`, and `RunningMaze` should be exclusive. The algorithms executed by the simulator are also exclusive. The behavior implemented in `UnderDebugging` should be applicable only when the system is in debug mode. It is difficult for meta-programming to mechanically check such properties. On the other hand, by using COP, we can easily generate a state transition model from the event-driven layer switching to perform model checking [21]. The exclusiveness of algorithms can be checked by checking only the exclusiveness of the simulator states that affect the value of the expressions used in the `if` expressions (e.g., the value of the `isRightHandRule` call). The dependency between `UnderDebugging` and `Debugging` is obtained immediately from the context specification of the activate declaration.

Finally, COP supports modularization of the specification that determines when behavior changes occur. If we apply other approaches to implement such behavior changes (e.g., the state design pattern), the behavior changes may be hardwired and scattered in the base program. Using the declarative specification of layer switching in COP languages, e.g., JCop [8] and EventCJ [21], such behavior changes are specified separately. Although the examples shown in this paper are written using imperative events for brevity, ServalCJ also supports declarative events using AspectJ-like pointcut language.

We further compared ServalCJ with existing COP languages. The case study showed that different activation mechanisms can be used in the same application. As discussed in Section 2.3, no existing COP language supports such a variety of activation mechanisms. There are no existing COP languages that support all event-based, per-control-flow, and implicit activation mechanisms, while ServalCJ supports all activation mechanisms (the imperative activation in Subjective-C can also be represented by from-to expressions where the until clause specifies an event that will never happen). Furthermore, the case study demonstrated how several *combinations* of activation mechanisms and sets of subscribers are used in the same application. In particular, the combination of global and per-control-flow activation and per-instance and implicit activation are used in the application. As Table 1 summarizes, existing COP languages do not support such combinations. Even combining these languages, where we can apply workarounds to represent such combinations, does not provide a sufficient solution. For example, when combining Subjective-C and ContextJ, the imperative activation can be used to globally activate and deactivate some layer `L` at the beginning and end of a `with`-block, respectively. In this workaround, it is the programmer's responsibility not to forget the deactivation of `L`. The errors caused by forgetting this deactivation can be avoided in ServalCJ by declaring a `cflow` in a global context group. Furthermore, ServalCJ provides a more expressive mechanism for representing per-instance and event-based activation than existing languages. In ServalCJ, there are no limitations for objects to dy-

namically subscribe to the context group, and we can specify the sender of the event.

7 Formal Model of Layer Activation

In this section, we formalize the semantics of layer activation in our model to precisely describe how generalized layer activation discussed in Section 4 is performed. We present a formal calculus Featherweight ServalCJ (FSCJ), which unifies different layer activation mechanisms (synchronous and asynchronous activation mechanisms and global and per-instance activation mechanisms) by combining two existing COP calculi, i.e., Featherweight EventCJ (FECJ) [3], which provides per-instance and asynchronous activation, and ContextFJ [18], which provides global and synchronous activation.

To prove that the activation policy discussed in Section 4 is always satisfied in FSCJ, we formally define the priority of a layer and describe a safety property, which we refer to as *priority preservation*. Intuitively, we consider that the priority of layer $\texttt{L}_1$ is higher than that of layer $\texttt{L}_2$ at the invocation of method `C.m` if (1) both $\texttt{L}_1$ and $\texttt{L}_2$ override `C.m` and (2) if the body of $\texttt{L}_2\texttt{.C.m}$ is executed, there must be an execution of $\texttt{L}_1\texttt{.C.m}$ prior to the execution of $\texttt{L}_2\texttt{.C.m}$. The priority preservation ensures that, if there are multiple layer activation mechanisms (e.g., synchronous and asynchronous) and some policy is defined between them (e.g., synchronously activated layers always have higher priority than asynchronously activated layers), this policy is satisfied during computation.

Based on these definitions, we prove priority preservation in FSCJ. Furthermore, we show that these definitions are not specific to FSCJ but are useful for discussing layer priority in other calculi for COP such as ContextFJ and context holders [4]. We believe that these definitions are also applicable to discuss the priorities of layers activated by the imperative activation mechanism and the dynamic scoping mechanism in ContextJS [24].

We also show that FSCJ is parameterized over the activation policies. The activation policy determined in Section 4 is application specific, and it is desirable for programmers to configure it for specific cases. FSCJ allows activation policies to be switched by switching only the definitions of some auxiliary functions without changing the reduction rules. The proof of priority preservation for different policies is obtained in a straightforward manner.

7.1 The Calculus

Syntax. The abstract syntax of FSCJ is shown in Fig. 12. Metavariable `C` ranges over class names; `L` ranges over layer names; `f` ranges over field names; `m` ranges over method names; ℓ ranges over labels, which include an empty label ϵ; ι ranges over instance labels; γ ranges over global labels; `v` and `w` range over values; and `x` ranges over variables, which include a special variable `this`. Overlines denote sequences: e.g., $\overline{\texttt{f}}$ stands for a possibly empty sequence $\texttt{f}_1, \cdots, \texttt{f}_n$. We also abbreviate a sequence of pairs by writing "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" for "$\texttt{C}_1\ \texttt{f}_1, \cdots, \texttt{C}_n\ \texttt{f}_n$," where

$$
\begin{array}{lll}
\texttt{CL} & ::= \texttt{class C} \lhd \texttt{C \{ } \overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\texttt{ \}} & (\textit{classes}) \\
\texttt{K} & ::= & (\textit{constructors}) \\
 & \quad \texttt{C(}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}} & \\
\texttt{M} & ::= \texttt{C m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{)\{ return e; \}} & (\textit{methods}) \\
\texttt{e},\texttt{d} & ::= \texttt{x} \mid \texttt{e}^{\ell}\texttt{.f} \mid \texttt{e}^{\ell}\texttt{.m(}\overline{\texttt{e}}^{\overline{\ell}}\texttt{)} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)} & (\textit{expressions}) \\
 & \quad \mid \texttt{proceed(}\overline{\texttt{e}}\texttt{)} \mid \texttt{with L e} & \\
 & \quad \mid \texttt{v} \mid \texttt{v<C,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{v}}\texttt{)} \mid \texttt{\{e\}} & \\
\texttt{t} & ::= \uparrow \texttt{L} \mid \downarrow \texttt{L} & (\textit{activation rules}) \\
\ell & ::= \iota \mid \gamma & (\textit{event labels}) \\
\texttt{p} & ::= \texttt{v} \mapsto \texttt{new C(}\overline{\texttt{v}}\texttt{)<}\overline{\texttt{L}}\texttt{>} & (\textit{partial stores}) \\
\mu & ::= \overline{\texttt{p}} & (\textit{stores}) \\
st & ::= \cdot \mid st \gg \overline{\texttt{L}} & (\textit{stack})
\end{array}
$$

Fig. 12. FSCJ: abstract syntax

n denotes the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$. Similarly, we write "$\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}$" as shorthand for the sequence of declarations "$\texttt{C}_1\ \texttt{f}_1\texttt{;} \ldots \texttt{C}_n\ \texttt{f}_n\texttt{;}$" and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;} \ldots \texttt{;this.f}_n\texttt{=f}_n\texttt{;}$". We use commas and semicolons for concatenations. We abbreviate a concatenation $\overline{\texttt{L}}_A;\overline{\texttt{L}}_S$ of asynchronously activated layers $\overline{\texttt{L}}_A$ and synchronously activated layers $\overline{\texttt{L}}_S$ simply as a sequence of layers $\overline{\texttt{L}}$ when such distinction is not important. It is assumed that sequences of field declarations, parameter names, layer names, and method declarations contain no duplicate names.

A class declaration `CL` consists of its name, its superclass name, field declarations $\overline{\texttt{C}}\ \overline{\texttt{f}}$, a constructor `K`, and method definitions $\overline{\texttt{M}}$. A constructor `K` is trivial; it only sets the initial values to the corresponding fields. A method `M` takes arguments $\overline{\texttt{x}}$ and returns the value of expression `e`. An expression can be a variable, field access, method invocation, object instantiation, synchronous layer activation `with`, `proceed` call, and special runtime expressions, such as a location `v`, `{e}`, and $\texttt{v<C,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{v}}\texttt{)}$. These runtime expressions are explained in the following. Note that FSCJ is a functional calculus; thus, all constructs (including `with`) return values.

A value `v` is a location. A store μ is a sequence of pairs of a location and an object. We write this pair as $\texttt{v} \mapsto \texttt{new C(}\overline{\texttt{v}}\texttt{)<}\overline{\texttt{L}}\texttt{>}$, which is read as "object $\texttt{new C(}\overline{\texttt{v}}\texttt{)<}\overline{\texttt{L}}\texttt{>}$ is stored at location `v`." This store is used to destructively update the set of layers associated with each object during computation. The runtime expression `{e}` appears only as a subterm of `with` under reduction. A stack st remembers a sequence of layers $\overline{\texttt{L}}$ before the reduction of `with` starts so that the computation can restore that sequence after it finishes the reduction of `with`. The runtime expression $\texttt{new C(}\overline{\texttt{v}}\texttt{)<C,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{e}}\texttt{)}$, where $\overline{\texttt{L}}'$ is assumed to be a prefix of $\overline{\texttt{L}}$, means that `m` is going to be invoked on $\texttt{new C(}\overline{\texttt{v}}\texttt{)}$. The annotation $\texttt{<C,}\overline{\texttt{L}}'\texttt{,}\overline{\texttt{L}}\texttt{>}$ indicates the cursor where method lookup should start. As explained in the following, this form allow us to give the semantics of `proceed` by simple substitution-based reduction.

A label attached to an expression denotes an event receiver that simplifies the asynchronous layer activation; $\texttt{e}^{\iota}$ represents a situation in which an event (that activates some layer) is received by `e`, and $\texttt{e}^{\gamma}$ represents a situation in which an event that globally activates some layer is sent by `e`. To represent an expression that does not receive or send any events, we introduce an empty label ϵ. Typically, we write `e` to mean $\texttt{e}^{\epsilon}$.

As in ContextFJ, the calculus does not provide syntax for layers because the syntactical details of layers, such as the difference between class-in-layer and layer-in-class styles [5], are not relevant. Partial methods are registered in a partial method table PT that maps a triple `C`, `L`, and `m` of class, layer, and method names, respectively, to a method definition. The calculus also provides an activation rule table TT that maps a label to an activation rule that is either an activation $\uparrow$ `L` (activating `L`) or a deactivation $\downarrow$ `L` (deactivating `L`). Note that $TT(\epsilon) = \emptyset$ (no layers are activated and deactivated).

FSCJ supports the multiple layer activation mechanisms described in Section 5.2. Intuitively, asynchronous layer activation triggered by the activation rules in TT corresponds to layer activation using conditionals and from-to expressions in ServalCJ. Synchronous layer activation represented by the `with` expressions in FSCJ corresponds to layer activation using `cflow` in ServalCJ. A value with a label $\texttt{v}^{\ell}$ corresponds to a subscriber in ServalCJ, and the global label γ indicates the global layer activation.

A program $(CT, PT, TT, \texttt{e})$ consists of a class table CT (that maps a class name to a class definition), a partial method table PT, an activation rule table TT, and an expression `e` that corresponds to the body of the main method. We assume CT, PT, and TT are fixed and satisfy the following sanity conditions:

1. $CT(\texttt{C}) = \texttt{class C ...}$ for any $\texttt{C} \in dom(CT)$.
2. $\texttt{Object} \notin dom(CT)$.
3. For every class name `C` (except `Object`) appearing anywhere in CT, we have $\texttt{C} \in dom(CT)$.
4. There are no cycles in the transitive closure of $\triangleleft$ (`extends`).
5. $PT(\texttt{m}, \texttt{C}, \texttt{L}) = \texttt{... m(...)\{...\}}$ for any $(\texttt{m}, \texttt{C}, \texttt{L}) \in dom(PT)$.
6. $TT(\ell) = \overline{\texttt{t}}$ for every label ℓ that appears in `e`, CT, and PT.

Auxiliary Functions. The operational semantics of FSCJ use auxiliary functions to look up field and method definitions. These lookup functions are defined in Fig. 13. The function *fields*(`C`) returns a sequence $\overline{\texttt{C}}\ \overline{\texttt{f}}$ of pairs of a field name and its type declared in class `C` and its superclasses. The function *mbody*(`m`,`C`,$\overline{\texttt{L}}_1$,$\overline{\texttt{L}}_2$) returns a pair $\overline{\texttt{x}}.\texttt{e}$ of parameters and the body of method `m` in class `C` when the search starts from $\overline{\texttt{L}}_1$. The other layer names $\overline{\texttt{L}}_2$ keep track of the layers that are active when the search initially started. It also returns the information where the method has been found. This information will be used in the reduction rules to handle `proceed`. The method definition is searched for in class `C` in all activated layers and then in the base definition. If no method definition is found, then the search continues to `C`'s superclass. Note that in MB-Super, which shows a case whereby the search proceeds to `C`'s superclass `D`, $\overline{\texttt{L}}$

$\boxed{\mathit{fields}(\mathtt{C})=\overline{\mathtt{C}}\ \overline{\mathtt{f}}}$

$$\mathit{fields}(\mathtt{Object}) = \bullet \qquad \text{(F-Object)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \ldots\ \} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}} \qquad \text{(F-Class)}$$

$\boxed{\mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}',\overline{\mathtt{L}})=\overline{\mathtt{x}}\mathtt{.e\ in\ D},\overline{\mathtt{L}}''}$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \ldots\ \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}\ \ldots\ \}}{\mathit{mbody}(\mathtt{m},\mathtt{C},\bullet,\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ C},\bullet} \qquad \text{(MB-Class)}$$

$$\frac{PT(\mathtt{m},\mathtt{C},\mathtt{L}_0) = \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}}{\mathit{mbody}(\mathtt{m},\mathtt{C},(\overline{\mathtt{L}}';\mathtt{L}_0),\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ C},(\overline{\mathtt{L}}';\mathtt{L}_0)} \qquad \text{(MB-Layer)}$$

$$\frac{PT(\mathtt{m},\mathtt{C},\mathtt{L}_0)\ \text{undefined} \qquad \mathit{mbody}(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}',\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ D},\overline{\mathtt{L}}''}{\mathit{mbody}(\mathtt{m},\mathtt{C},(\overline{\mathtt{L}}';\mathtt{L}_0),\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ D},\overline{\mathtt{L}}''} \qquad \text{(MB-NextLayer)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \ldots\ \overline{\mathtt{M}}\ \} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathit{mbody}(\mathtt{m},\mathtt{D},\overline{\mathtt{L}},\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ E},\overline{\mathtt{L}}'}{\mathit{mbody}(\mathtt{m},\mathtt{C},\bullet,\overline{\mathtt{L}}) = \overline{\mathtt{x}}\mathtt{.e\ in\ E},\overline{\mathtt{L}}'} \qquad \text{(MB-Super)}$$

Fig. 13. FSCJ: lookup functions

is copied to the third argument in the premise in order to consider all activated layers.

Operational Semantics. The operational semantics are given by a reduction relation of the form $\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{e}' \mid \mu' \mid \overline{\mathtt{L}}' \mid st'$, which is read as "expression $\mathtt{e}$ under a store μ, globally activated layers $\overline{\mathtt{L}}$, and a stack st reduces to $\mathtt{e}'$ under μ', $\overline{\mathtt{L}}'$, and st'." We assume that neither μ nor μ' contain duplicate names.

The reduction rules for layer activation and deactivation are shown in Fig. 14. The rule R-LabelActI represents the reduction that occurs when a value $\mathtt{v}$ receives an event denoted by label ι. This rule obtains the corresponding layer activation rule stored in $TT$and calculates the order of active layers by applying *actAsync*. The store μ is updated by inserting the location of the instance with new active layers. The layer deactivation is provided by R-LabelDeactI, which is obtained by replacing $\uparrow$ and *actAsync* in R-LabelActI with $\downarrow$ and *deact*, respectively.

Similarly, the rule R-LabelActG represents the reduction that occurs when a value $\mathtt{v}$ sends an event denoted by label γ, which triggers the global layer

$$\boxed{\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{e}' \mid \mu' \mid \overline{\mathtt{L}}' \mid st'}$$

$$\frac{\begin{array}{cc} TT(\iota) = \uparrow \mathtt{L} & \mu(\mathtt{v}) = \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}'\mathtt{>} \\ actAsync(\overline{\mathtt{L}}', \mathtt{L}) = \overline{\mathtt{L}}'' & \mu' = (\mathtt{v} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}''\mathtt{>}, \mu) \end{array}}{\mathtt{v}^{\iota} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{v} \mid \mu' \mid \overline{\mathtt{L}} \mid st} \quad \text{(R-LabelActI)}$$

$$\frac{\begin{array}{cc} TT(\iota) = \downarrow \mathtt{L} & \mu(\mathtt{v}) = \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}'\mathtt{>} \\ deact(\overline{\mathtt{L}}', \mathtt{L}) = \overline{\mathtt{L}}'' & \mu' = (\mathtt{v} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}''\mathtt{>}, \mu) \end{array}}{\mathtt{v}^{\iota} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{v} \mid \mu' \mid \overline{\mathtt{L}} \mid st} \quad \text{(R-LabelDeactI)}$$

$$\frac{\begin{array}{cc} TT(\gamma) = \uparrow \mathtt{L} & actAsync_{\mu}(\mu, \mathtt{L}) = \mu' \\ actAsync(\overline{\mathtt{L}}, \mathtt{L}) = \overline{\mathtt{L}}' & actAsync_{st}(st, \mathtt{L}) = st' \end{array}}{\mathtt{v}^{\gamma} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{v} \mid \mu' \mid \overline{\mathtt{L}}' \mid st'} \quad \text{(R-LabelActG)}$$

$$\frac{\begin{array}{cc} TT(\gamma) = \downarrow \mathtt{L} & deact_{\mu}(\mu, \mathtt{L}) = \mu' \\ deact(\overline{\mathtt{L}}, \mathtt{L}) = \overline{\mathtt{L}}' & deact_{st}(st, \mathtt{L}) = st' \end{array}}{\mathtt{v}^{\gamma} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{v} \mid \mu' \mid \overline{\mathtt{L}}' \mid st'} \quad \text{(R-LabelDeactG)}$$

$$\frac{actSync(\overline{\mathtt{L}}, \mathtt{L}) = \overline{\mathtt{L}}'}{\mathtt{with\ L\ e} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \{\mathtt{e}\} \mid \mu \mid \overline{\mathtt{L}}' \mid st \gg \overline{\mathtt{L}}} \quad \text{(R-ActSync)}$$

$$\frac{\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{e}' \mid \mu' \mid \overline{\mathtt{L}}' \mid st'}{\{\mathtt{e}\} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \{\mathtt{e}'\} \mid \mu' \mid \overline{\mathtt{L}}' \mid st'} \quad \text{(R-ActSyncCont)}$$

$$\{\mathtt{v}\} \mid \mu \mid \overline{\mathtt{L}} \mid st \gg \overline{\mathtt{L}}' \longrightarrow \mathtt{v} \mid \mu \mid \overline{\mathtt{L}}' \mid st \quad \text{(R-ActSyncFin)}$$

Fig. 14. FSCJ: reduction rules (1)

activation. This rule updates the sequence of globally activated layers $\overline{\mathtt{L}}$. It also applies *actAsync* to all the elements in μ using the auxiliary function $actAsync_{\mu}$, which is defined as follows.

$$\frac{\mu' = \{\mathtt{v} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}'\mathtt{>} \mid \mathtt{v} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}\mathtt{>} \in \mu, \overline{\mathtt{L}}' = actAsync(\overline{\mathtt{L}}, \mathtt{L})\}}{actAsync_{\mu}(\overline{\mathtt{L}}, \mathtt{L}) = \mu'}$$

i.e., for each sequence of active layers $\overline{\mathtt{L}}_i$ in the range of μ is updated by applying $actAsync(\overline{\mathtt{L}}_i, \mathtt{L})$. Similarly, R-LabelActG applies *actAsync* to the stack *st* using the following auxiliary function.

$$actAsync_{st}(\cdot) = \cdot$$

$$\frac{actAsync_{st}(st) = st' \qquad actAsync(\overline{\mathtt{L}}) = \overline{\mathtt{L}}'}{actAsync_{st}(st \gg \overline{\mathtt{L}}) = st' \gg \overline{\mathtt{L}}'}$$

The rule R-LabelDeactG defines the global layer deactivation. This rule is obtained by replacing *actAsync* in R-LabelActG with *deact*. The auxiliary functions $deact_{\mu}$ and $deact_{st}$ are defined as follows.

$$\frac{\mu' = \{\mathtt{v} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}'\mathtt{>} \mid \mathtt{v} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}\mathtt{>} \in \mu, \overline{\mathtt{L}}' = deact(\overline{\mathtt{L}}, \mathtt{L})\}}{deact_{\mu}(\overline{\mathtt{L}}, \mathtt{L}) = \mu'}$$

$$deact_{st}(\cdot) = \cdot$$

$$\frac{deact_{st}(st) = st' \qquad deact(\overline{\mathtt{L}}) = \overline{\mathtt{L}}'}{deact_{st}(st \gg \overline{\mathtt{L}}) = st' \gg \overline{\mathtt{L}}'}$$

Note that there is a subtle problem here; that the global layer deactivation can deactivate a layer that has been activated by `with`, which may contradict our proposal that synchronous layer activation dominates asynchronous layer activation, because, in FSCJ (and in the implementation of ServalCJ), there is no runtime information that retains the synchronously activated layers (the stack does not remember the layer activated by synchronous activation but remembers the layers activated prior to synchronous activation). This problem can be resolved if we manage the synchronous layer activation separately, similar to context holders [4].

The R-ActSync rule defines synchronous layer activation. The *actSync* function places `L` on top of the sequence of activated layers $\overline{\mathtt{L}}'$ and ensures it is activated during the evaluation of body `e`, which is reduced to the runtime expression {`e`}. Stack *st* is updated so that it can pop $\overline{\mathtt{L}}$, i.e., the globally activated layers before the evaluation of `with`, when the evaluation of the body `e` is finished. The reduction rules for this runtime expression are given as rules R-ActSyncCont and R-ActSyncFin, and the R-ActSyncFin rule ensures that the pop operation can be applied only when the expression is in the form of {`v`}, i.e., it preserves the push-pop correspondence in the nested `with`-blocks.

The reduction rules for field access, method invocation, and instance creation are shown in Fig. 15. The rule R-Field for field access is straightforward, i.e., *fields* tells which argument to `new C(..)` corresponds to $\mathtt{f}_i$. The next three rules are for method invocation. In the method lookup, we must include layers that are activated globally and synchronously in the search sequence, as shown in the R-Invk, R-InvkB and R-InvkP rules. The cursor for the method lookup is set as a concatenation of the asynchronously activated layers in a per-instance manner $\overline{\mathtt{L}}''_A$ and globally and synchronously activated layers $\overline{\mathtt{L}}_S$. The auxiliary definition *o* represents concatenation of the asynchronously activated layers and synchronously activated layers, and is defined as follows.

$$o(\overline{\mathtt{L}}_A, \overline{\mathtt{L}}_S) = \overline{\mathtt{L}}_A; \overline{\mathtt{L}}_S$$

$$\boxed{\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{e}' \mid \mu' \mid \overline{\mathtt{L}}' \mid st'}$$

$$\frac{\mu(\mathtt{v}) = \mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{)<}\overline{\mathtt{L}}'\mathtt{>} \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{v.f}_i \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{w}_i \mid \mu \mid \overline{\mathtt{L}} \mid st} \quad \text{(R-Field)}$$

$$\frac{\mu(\mathtt{v}_0) = \mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{)<}\overline{\mathtt{L}}''_A\mathtt{>} \qquad \overline{\mathtt{L}}'' = o(\overline{\mathtt{L}}''_A, \overline{\mathtt{L}}_S) \qquad \overline{\mathtt{L}} = o(\overline{\mathtt{L}}_A, \overline{\mathtt{L}}_S) \qquad \mathtt{v}_0\mathtt{<C,}\overline{\mathtt{L}}''\mathtt{>.m(}\overline{\mathtt{v}}\mathtt{)} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{e} \mid \mu' \mid \overline{\mathtt{L}}' \mid st'}{\mathtt{v}_0\mathtt{.m(}\overline{\mathtt{v}}\mathtt{)} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{e} \mid \mu' \mid \overline{\mathtt{L}}' \mid st'} \quad \text{(R-Invk)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}\mathtt{.e\ in\ C}', \bullet}{\mathtt{v<C,}\overline{\mathtt{L}}''\mathtt{,}\overline{\mathtt{L}}'\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow [\mathtt{v/this}, \overline{\mathtt{w}}/\overline{\mathtt{x}}]\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \mid st} \quad \text{(R-InvkB)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}'', \overline{\mathtt{L}}') = \overline{\mathtt{x}}\mathtt{.e\ in\ C}', (\overline{\mathtt{L}}'''; \mathtt{L}_0)}{\mathtt{v<C,}\overline{\mathtt{L}}''\mathtt{,}\overline{\mathtt{L}}'\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow [\mathtt{v/this}, \overline{\mathtt{w}}/\overline{\mathtt{x}}, \mathtt{v<C'},\overline{\mathtt{L}}'''\mathtt{,}\overline{\mathtt{L}}'\mathtt{>.m/proceed}]\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \mid st} \quad \text{(R-InvkP)}$$

$$\frac{\mathtt{w} \notin \mathit{dom}(\mu) \qquad \overline{\mathtt{L}} = o(\overline{\mathtt{L}}_A, \overline{\mathtt{L}}_S)}{\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{w} \mid (\mathtt{w} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}_A\mathtt{>}, \mu) \mid \overline{\mathtt{L}} \mid st} \quad \text{(R-New)}$$

Fig. 15. FSCJ: reduction rules (2)

This definition, along with *actSync* and *actAsync*, can be parameterized. As discussed in Section 7.4, giving different definitions for them results in another calculus that conforms to another activation policy, e.g., asynchronously activated layers always supersede synchronously activated layers.

The R-New rule explains the reduction of an instance creation. Note that globally and asynchronously activated layers $\overline{\mathtt{L}}_A$ are prospectively activated in the new instance. Also note that each instance $\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}\mathtt{>}$ only has asynchronously activated layers.

Finally, we provide a straightforward congruence rule that reduces a subexpression with a label.

$$\frac{\mathtt{e} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \mathtt{e}' \mid \mu' \mid \overline{\mathtt{L}}' \mid st'}{G[\mathtt{e}^{\ell}] \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow G[\mathtt{e}'^{\ell}] \mid \mu' \mid \overline{\mathtt{L}}' \mid st'} \quad \text{(RC-Label)}$$

$G[\cdot]$ forms evaluation contexts, which is defined as follows.

$$G ::= [] \mid G\mathtt{.m(}\overline{\mathtt{e}}^{\ell}\mathtt{)} \mid \mathtt{v}^{\ell}\mathtt{.m(}\overline{\mathtt{w}}^{\ell}\mathtt{,}G\mathtt{,}\overline{\mathtt{e}}^{\ell}\mathtt{)} \mid G\mathtt{.f} \mid \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{,}G\mathtt{,}\overline{\mathtt{e}}\mathtt{)}$$

We write $G[\mathtt{e}^{\ell}]$ for the ordinary expression obtained by replacing the hole in G with $\mathtt{e}^{\ell}$. As in FECJ, FSCJ requires the receiver and all arguments on the left of the redex to be values.

Example. In the piece of code in Section 3.2, the layers activated by synchronous layer activation are pushed to list $\bar{\mathtt{L}}_S$, and the layer activated by an event is pushed to list $\bar{\mathtt{L}}_A$. Thus, the resulting order of the active layers is as follows.

```
EditingComments;EditingCode,RenderingCode
```

Thus, the priority of the `EditingCode` layer is higher than that of the `EditingComments` layer, ensuring that syntax highlighting is always applied when the `sh.format(textBlock)` method call is executed.

Since FSCJ does not provide layer-introduced base methods (i.e., methods defined in layers but not defined in base classes) [19], the type system of FSCJ is trivial. It is a straightforward adaptation of the type system of (an earlier version) of ContextFJ [18]. The type soundness of FSCJ is also obtained by straightforward adaptation of the proof of type soundness in ContextFJ, which is also a straightforward adaptation of the proof of type soundness in Featherweight Java [20]. In this paper, we omit the type system and type soundness for simplicity.

7.2 Definition of Priority

In the previous example, the priorities of layers are as expected with respect to the activation policy discussed in Section 4. A formal study is a promising approach to ensure that this activation policy is preserved during computation.

In this section, we formally provide a definition of layer priority. To make the definition general and language-independent, we do not define priority in terms of language-specific features, such as the ordering of activated layers. Instead, we consider the history of activation and deactivation to define priority. To describe such a history, we first define a trace of the execution. Let $\mathtt{e}_0 \longrightarrow^* \mathtt{e}_n$ be the transitive closure of the smallstep reduction of some COP calculus.[9] A trace t is a sequence $\mathtt{e}'_0|\Lambda_0, \cdots, \mathtt{e}'_n|\Lambda_n$ of a pair of an expression and a set of activated layers[10], where each $\mathtt{e}'_i$ is the redex in $\mathtt{e}_i$ replaced with a subexpression in $\mathtt{e}_{i+1}$ in one reduction step. Each Λ_i is a set of activated layers of a particular computation unit on which we focus when $\mathtt{e}_i$ is to be evaluated. For example, for a *trace of the application* in ContextFJ, each Λ_i is a set of activated layers in the runtime environment where $\mathtt{e}_i$ is to be evaluated. Similarly, for a *trace of value* $\mathtt{v}$ in FSCJ, each Λ_i is a set of activated layers associated with $\mu(\mathtt{v})$ in the runtime environment where $\mathtt{e}_i$ is to be evaluated. We also define an activation sequence for t, written $act(t)$, which is a sequence $\alpha_0, \cdots, \alpha_{n-1}$, where each α_i

[9] While we consider the definitions that are independent from the activation mechanisms, we still assume that layers and constructs of the host language are based on ContextFJ-like calculi, e.g., we assume the existence of *mbody* and substitution-based reduction for `proceed`.

[10] To speak of the layer priority, we focus on *which partial method executes first* rather than the ordering of the activated layers.

is either ϕ (no layers are activated and deactivated), $\uparrow$ L (L is activated), or $\downarrow$ L (L is deactivated). We assume that a trace t satisfies the following conditions.

$$\begin{array}{ll} \Lambda_i = \Lambda_{i+1} & \text{if } \alpha_i = \phi, \alpha_i \in act(t) \\ \Lambda_i \cup \{\mathtt{L}\} = \Lambda_{i+1} & \text{if } \alpha_i = \uparrow \mathtt{L}, \alpha_i \in act(t) \\ \Lambda_i \setminus \{\mathtt{L}\} = \Lambda_{i+1} & \text{if } \alpha_i = \downarrow \mathtt{L}, \alpha_i \in act(t) \end{array}$$

In other words, at most one layer is added to (removed from) Λ_i to obtain Λ_{i+1}, and each α_i is constructed by taking the difference between Λ_i and Λ_{i+1}. Note that each *act* does not correspond to each activation operation in the reduction steps. It represents an observed activation when focusing on the set of activated layers. For example, an activation operation that does not change the set of activated layers (i.e., an operation activating an already activated layer) is not captured in the trace.

Example. Let $(CT, PT, TT, \texttt{new C()}^{\gamma}\texttt{.m(with L' new C())})$ be a well-typed FSCJ program where $TT(\gamma) = \uparrow$ L. We have the following reduction steps for this program (each subexpression with an underline is replaced with another expression in each reduction step).

$$\begin{array}{ll} \underline{\texttt{new C()}^{\gamma}}\texttt{.m(with L' new C())} & \\ \quad \longrightarrow \underline{\texttt{v}^{\gamma}}\texttt{.m(with L' new C())} & \text{where } \mu(\texttt{v}) = \texttt{new C()} \\ \quad \longrightarrow \texttt{v.m(}\underline{\texttt{with L' new C()}}\texttt{)} & \\ \quad \longrightarrow \texttt{v.m(\{ }\underline{\texttt{new C()}}\texttt{ \})} & \\ \quad \longrightarrow \texttt{v.m(\{ }\underline{\texttt{v'}}\texttt{ \})} & \text{where } \mu(\texttt{v}') = \texttt{new C()} \\ \quad \longrightarrow \underline{\texttt{v.m(v')}} & \\ \quad \longrightarrow \cdots & \end{array}$$

By listing each underlined subexpression, we have the trace t of the value v and its activation sequence as follows.[11]

$$\begin{array}{rl} t = & \texttt{new C()}^{\gamma} \mid \emptyset, \texttt{v}^{\gamma} \mid \emptyset, \texttt{with L' new C()} \mid \{\mathtt{L}\}, \texttt{new C()} \mid \{\mathtt{L}, \mathtt{L}'\}, \\ & \{\texttt{v}'\} \mid \{\mathtt{L}, \mathtt{L}'\}, \texttt{v.m(v')} \mid \{\mathtt{L}\}, \cdots \\ act(t) = & \phi, \uparrow \mathtt{L}, \uparrow \mathtt{L}', \phi, \downarrow \mathtt{L}, \phi, \cdots \end{array}$$

A trace contains a history of layer activation, as well as a history of field accesses and method invocations, including partial method invocations (of the form v<C,$\overline{\mathtt{L}}$, $\overline{\mathtt{L}}'$>.m(..)) for each value. We can define the relation between layers at an execution point, i.e., the relation by which "the priority of layer L is higher than the priority of L' at $\mathtt{e}_m$," in terms of the positions of the partial method invocations in the trace. Note that, if multiple occurrences of the same value v exist in the expression $\mathtt{e}_i$ in the reduction steps, each such value is uniquely renamed to make each value identical in the trace. This renaming prevents us from mixing multiple calls of the same method in the same expression. For example, assuming the expression v<C,$\overline{\mathtt{L}}$,$\overline{\mathtt{L}}'$>.m(v.m(v')), the underlined expression

[11] All Λ_i are empty before v is created.

is reduced to $\texttt{v<C,}\overline{\texttt{L}}''\texttt{,}\overline{\texttt{L}}''\texttt{>.m(..)} = \texttt{e}_{m'}$, which is added to the trace. Then, it becomes difficult to determine which method call, i.e., the outer or inner call of `m`, was evaluated to produce $\texttt{e}_{m'}$. To avoid this, the inner `v` is renamed as another value, e.g., `v1`, in the trace.

Definition 1. *Suppose we have a trace* $t = \texttt{e}_0 \mid \Lambda_0, \cdots, \texttt{e}_n \mid \Lambda_n$ *and an expression* $\texttt{e}_m = \texttt{v.m(..)}$ *in* t *where* `v` *is a value of type* `C`. *Assume that* $\texttt{L}, \texttt{L}' \in \Lambda_m$ *and both* `L` *and* $\texttt{L}'$ *provide a partial method* `m` *that overrides the method* `m` *in* `C`. *We say that* `L`*'s priority is higher than the priority of* $\texttt{L}'$ *at* $\texttt{e}_m$ *iff, if* $\mathit{mbody}(\texttt{m}, \texttt{C}, \cdots) = \overline{\texttt{y}}\texttt{.e}'\ \texttt{in D}, (\cdots; \texttt{L}')$ *for some* $\overline{\texttt{y}}$, $\texttt{e}'$, *and* `D` *on the reduction of* $\texttt{v<C,..>.m(..)} = \texttt{e}_{o'}$ *where* $m < o'$ *and* $\forall i'$ *such that* $m < i' < o', \alpha_{i'} \neq \uparrow \texttt{L}'$, *then there is some* o *such that* $m < o < o'$ *and* $\mathit{mbody}(\texttt{m}, \texttt{C}, \cdots) = \overline{\texttt{x}}\texttt{.e in E}, (\cdots; \texttt{L})$ *for some* $\overline{\texttt{x}}$, `e` *and* `E` *on the reduction of* $\texttt{v<C,..>.m(..)} = \texttt{e}_o$.

While existing COP calculi express the precedence of layers in terms of layer ordering, this definition of priority is applicable to other COP models that do not incorporate the notion of ordering (e.g., a (imaginary) COP model where layer priorities are defined statically). Of course, this definition of priority is applicable to existing COP calculi. For example, ContextFJ [18] applies the semantics that the most recently activated layer has the highest priority, which is expressed by the following example (we write $\{\overline{\texttt{L}}_i\}$ when we regard a sequence $\overline{\texttt{L}}$ as a set).

Example. Let $(CT, PT, \texttt{e}_0)$ be a well-typed ContextFJ program and $\bullet \vdash \texttt{e}_0 \longrightarrow^* \texttt{e}_n$ be a transitive closure of reduction steps in ContextFJ. Let $t = \texttt{e}_0 \mid \{\overline{\texttt{L}}_0\}, \cdots, \texttt{e}_n \mid \{\overline{\texttt{L}}_n\}$, where each $\overline{\texttt{L}}_i$ is an environment in which reduction of $\texttt{e}_i$ is performed, be a trace of the application in ContextFJ (rather than applying the original ContextFJ dynamic semantics, we apply FSCJ-like semantics; i.e., a `with` expression is reduced using R-ActSync in FSCJ), and both $\uparrow \texttt{L} = \alpha_i$ and $\uparrow \texttt{L}' = \alpha_j \in \mathit{act}(t)$ are the most recent activations of `L` and $\texttt{L}'$ from the method invocation $\texttt{e}_m = \texttt{v.m(..)}$, respectively. It is easy to show that, if $i < j$, the priority of `L` is higher than the priority of $\texttt{L}'$ at $\texttt{e}_m$.

In the calculus that supports multiple activation mechanisms, such as FSCJ and context holders [4], it is desirable to discuss the priorities of layers activated by different activation mechanisms. For this purpose, we distinguish the different categories of activation mechanisms, and extend the definition of a trace to express the activation sequence with multiple activation mechanisms. A trace t with multiple activation mechanisms $\texttt{X}_j$ $(1 < j < m)$ is a sequence $\texttt{e}_0 \mid \{\Lambda_{01}, \cdots, \Lambda_{0m}\}, \cdots, \texttt{e}_n \mid \{\Lambda_{n1}, \cdots, \Lambda_{nm}\}$ of a pair of an expression and a set of sets of activated layers where each Λ_{ij} is a set of activated layers at $\texttt{e}_i$ that are activated by $\texttt{X}_j$.

7.3 Property

We show that the activation policy discussed in Section 4 holds in FSCJ, which is represented by the following priority preservation theorem.

Theorem 1 (priority preservation). *For all* $\mathtt{e}_m = \mathtt{v.m(..)}$ *in a trace* $t = \mathtt{e}_0 \mid \{\{\overline{\mathtt{L}}_{S0}\}, \{\overline{\mathtt{L}}_{A0}\}\}, \cdots, \mathtt{e}_n \mid \{\{\overline{\mathtt{L}}_{Sn}\}, \{\overline{\mathtt{L}}_{An}\}\}$ *of* v *in FSCJ,* $\forall \mathtt{L}, \mathtt{L}'$ *where* $\mathtt{L} \in \{\overline{\mathtt{L}}_{Sm}\}$ *and* $\mathtt{L}' \in \{\overline{\mathtt{L}}_{Am}\}$, L*'s priority is higher than that of* $\mathtt{L}'$ *at* $\mathtt{e}_m$.

Proof. By the definitions of *actSync* and *actAsync*, and the fact that trace t is constructed from a transitive closure of reduction steps in FSCJ, there is a store μ in the runtime environment where an expression that has $\mathtt{e}_m$ as a subexpression to be evaluated, and $\mu(\mathtt{v}) = \mathtt{new\ C(..)<}\overline{\mathtt{L}}\mathtt{>}$ where $\overline{\mathtt{L}} = \overline{\mathtt{L}}_{Am}; \overline{\mathtt{L}}_{Sm}$. Since $\mathtt{L} \in \{\overline{\mathtt{L}}_{Sm}\}$ and $\mathtt{L}' \in \{\overline{\mathtt{L}}_{Am}\}$, we can also write $\overline{\mathtt{L}} = \overline{\mathtt{L}}'; \mathtt{L}'; \overline{\mathtt{L}}'''; \mathtt{L}; \overline{\mathtt{L}}''$. We prove this theorem by induction on the length of $\overline{\mathtt{L}}'''$.

Base case: $\overline{\mathtt{L}} = \overline{\mathtt{L}}'; \mathtt{L}'; \mathtt{L}; \overline{\mathtt{L}}''$.

By the definition of priority, we only consider the case where both $PT(\mathtt{m}, \mathtt{C}, \mathtt{L})$ and $PT(\mathtt{m}, \mathtt{C}, \mathtt{L}')$ are defined. Then, $mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}'; \mathtt{L}', \overline{\mathtt{L}}) = \overline{\mathtt{y}}.\mathtt{e}'\ \mathtt{in\ D}$, $(\overline{\mathtt{L}}; \mathtt{L}')$ for some $\overline{\mathtt{y}}$, $\mathtt{e}'$, and D. By the definition of *mbody* and R-InvkP, there must be $\mathtt{e}_p = \mathtt{v<C,}\overline{\mathtt{L}}_0\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(..)}$ for some $\overline{\mathtt{L}}_0 = \overline{\mathtt{L}}'; \mathtt{L}'; \cdots$ and $\overline{\mathtt{L}} = \overline{\mathtt{L}}_0; \cdots$ and $p < o'$ and $mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}_0, \overline{\mathtt{L}}) = \overline{\mathtt{y}}'.\mathtt{e}''\ \mathtt{in\ E}, (\cdots; \mathtt{L}'')$. Without loss of generality, we can let $\overline{\mathtt{L}}_0 = \overline{\mathtt{L}}'; \mathtt{L}'; \mathtt{L}$. Then, by the definition of *mbody*, $\mathtt{L}'' = \mathtt{L}$, finishes the case.

Case: $\overline{\mathtt{L}} = \overline{\mathtt{L}}'; \mathtt{L}'; \overline{\mathtt{L}}'''; \mathtt{L}; \overline{\mathtt{L}}''$.

Let $\overline{\mathtt{L}}''' = \overline{\mathtt{L}}_1; \mathtt{L}'''$.

By the hypothesis of the induction, the priority of $\mathtt{L}'''$ is higher than L'. The base case tells us that L's priority is higher than the priority of $\mathtt{L}'''$. It is obvious that this priority relation has transitivity; thus, L's priority is higher than $\mathtt{L}'$'s, which finishes the case. □

7.4 Changing the Activation Policy

In Section 4, we determined the activation policy, i.e., synchronously activated layers always have higher priorities than asynchronously activated layers. Although this policy seems preferable in many cases, there may be other cases in which asynchronous layer activation should supersede synchronous layer activation. For example, some urgent behavior triggered by an external event should supersede the currently executing synchronously activated behavior.

This section demonstrates that the activation policy in FSCJ is configurable. By changing the auxiliary definitions used in the reduction rules, we obtain a calculus that conforms to another activation policy, i.e., asynchronously activated layers always have higher priorities than synchronously activated layers, without changing the reduction rules.

The auxiliary functions that we need to change when switching the activation policies are *actSync* and *actAsync*, which are defined in Section 4.2. Assume that their definitions are overridden as follows.

$$actSync(\bar{\mathtt{L}}_S; \bar{\mathtt{L}}_A, \mathtt{L}) = \begin{cases} (\bar{\mathtt{L}}_S \setminus \mathtt{L})\mathtt{L}; \bar{\mathtt{L}}_A & \text{if } \mathtt{L} \notin \bar{\mathtt{L}}_A \\ \bar{\mathtt{L}}_S; \bar{\mathtt{L}}_A & \text{if } \mathtt{L} \in \bar{\mathtt{L}}_A \end{cases}$$

$$actAsync(\bar{\mathtt{L}}_S; \bar{\mathtt{L}}_A, \mathtt{L}) = (\bar{\mathtt{L}}_S \setminus \mathtt{L}); (\bar{\mathtt{L}}_A \setminus \mathtt{L})\mathtt{L}$$

We also override the auxiliary function o that is used in rules R-Invk and R-New.

$$o(\overline{\mathsf{L}}_A, \overline{\mathsf{L}}_S) = \overline{\mathsf{L}}_S; \overline{\mathsf{L}}_A$$

The calculus obtained by applying these auxiliary functions conforms to another activation policy. In fact, the proof is obtained immediately by switching $\{\overline{\mathsf{L}}_{Sm}\}$ and $\{\overline{\mathsf{L}}_{Am}\}$ in Theorem 1.

8 Implementation

The ServalCJ compiler is built on top of the AspectBench Compiler (abc) [9] by extending the front-end. The compiler eventually generates bytecode that is executable on the standard Java virtual machine by first translating a ServalCJ program into an AspectJ program, and then by having the AspectJ compiler generate bytecode.[12]

8.1 Overview of Translation

The translation processes manipulate the following four constructs separately: partial and base methods, conditional layer activation, global layer activation, and events. The main differences between the ServalCJ and EventCJ compilers are the implementations of layer activation using conditional expressions and global layer activation.

We first explain how layers are translated. The translation is similar to that performed in the EventCJ compiler [21]. A layer is translated into an inner class, and each partial method in that layer is translated into a method in that inner class. The body of the base method for that partial method is translated to code that first obtains the list of instances of active layers (i.e., instances of the inner classes) and then calls the instance method at the tail position of the list. The proceed call is translated to code that calls the method on the instance at the preceding position in the list. Every class extended by partial methods will have a new field, `lm`, to store a list of active layers.

The conditional expressions are evaluated just before a partial method call. The ServalCJ compiler inserts checking code at the beginning of the layered method. The checking code (1) tests whether the instance executing the method subscribes to some context groups and (2) collects a list of context groups where the instance subscribes. For each context group, the checking code evaluates a conditional expression (associated with that context group). If any of the conditions hold and the corresponding layer is inactive, the code activates the layer. On the other hand, if they do not hold and the corresponding layer is active, the code deactivates the layer.

ServalCJ implements global layer activation using the per-instance layer activation mechanism. It places globally active layers in the list of active layers in

[12] The source code of the compiler is available at `https://github.com/ServalCJ/pl.git`. Per-thread activation is currently not implemented.

every instance. To do so, the runtime manages a list of all instances in a program.[13] When a global layer is activated, that layer is added to the `lm` field of every instance in the list. The runtime also manages a list of globally activated layers (that correspond to the global active layers $\overline{\mathrm{L}}$ that appear in the reduction relation in Appendix 7). This is used as an initial value of the list of active layers for a newly created instance.

Events in ServalCJ are translated into pointcuts in AspectJ. As in EventCJ, for each join point, the compiler inserts the advice code to update the `lm` field of each subscriber and the list of globally activated layers to perform layer activation and deactivation. Cflow expressions are also implemented in a similar manner, except that, in this case the advice code counts the number of method calls to handle recursive calls appropriately. For layer activations with multiple contexts using the `&&` and `||` operators, the compiler resolves which layers should be active on each join-point specified by the pointcuts.

8.2 Microbenchmarks

In this section, we evaluate the performance of method dispatch in ServalCJ by comparing the duration of method calls with and without active layers in ServalCJ with the duration of method calls in plain Java. To evaluate the overhead imposed on the compiled program, we conducted two experiments. The first experiment was performed to verify that ServalCJ does not degrade the execution performance significantly when we do not use the ServalCJ specific features that impose additional overhead. The objective of the second experiment was to measure the overhead of layered method calls with implicit and global activations.

We used JGFMethodBench in the Java Grande Forum Benchmark Suite [11] version 2 as the benchmark. We extended this benchmark to evaluate the layered method. For example, each target method in the program was extended using an around partial method that contained only the `proceed` call. All experiments were performed using the Oracle Java HotSpot VM 1.7.0_65 running on an Intel Core i5-4440 (4 cores, 3.10 GHz) with Linux kernel version 2.6.32. To prohibit the JIT compiler from eliminating the entire target method call, which is always performed in the server VM (the default setting of the Java HotSpot VM) and prohibits measuring overhead with respect to method calls, we used the client VM setting. This avoids such elimination in the benchmark.

Fig. 16 summarizes the method dispatch time in Java and ServalCJ *without active layers*. The benchmark program measured the execution time of eight types of method calls. The labels "same" and "other" indicate that the caller and callee methods belong to the same or another instance/class, respectively. "Instance" indicates that the method is an instance method, and "class" indicates that it is a class method. "Synchronized" and "ofAbstract" indicate that

[13] Precisely, only instances that have globally activated layers are added to the list to reduce the performance degradation.

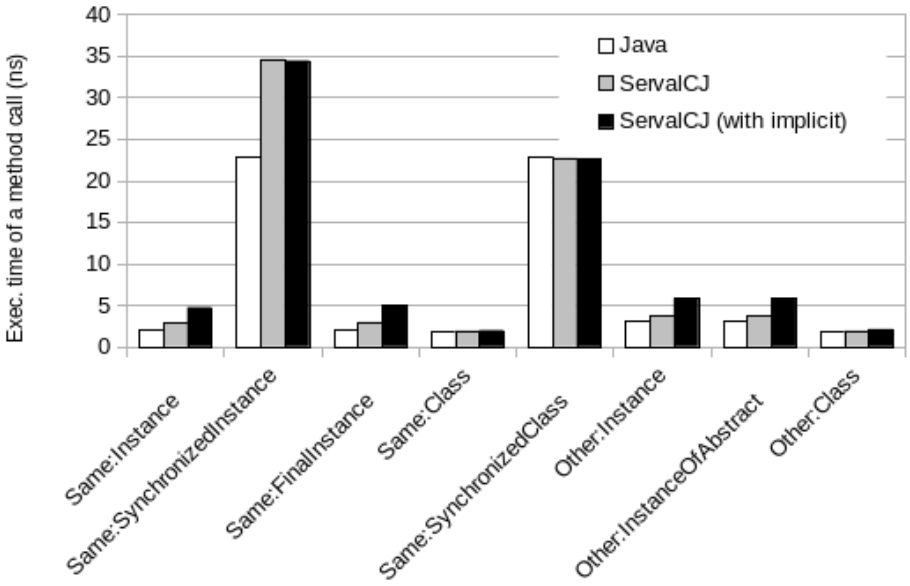


Fig. 16. Execution time of a method call in ServalCJ and Java (shorter is better). We ran the benchmarks 10 times; the range of error was approximately 0.007% – 1%.

the method is either synchronized or abstract, respectively. In the ServalCJ version, we defined a layer with a partial method for the instance methods that is inactive during measurement. We did not provide any partial methods for the class methods because ServalCJ does not currently support this.

Fig. 16 shows that, when no layers are active, the performance of method calls in ServalCJ is comparable to that of plain Java if implicit activation is not used. The primary reason for this is that, in this case, ServalCJ does not impose overhead on the program except, with the exception of the overhead incurred when it checks the number of currently active layers. The method call is approximately two times slower if we use implicit activation where the conditional expression in `if` is always evaluated just prior to the call of the partial method; this overhead is also comparable to other COP languages.

Fig. 17 shows the results of measuring the method dispatch time in ServalCJ *with 1 to 15 active layers*. In this experiment, we defined 15 identical layers, each of which declared an around partial method that contained only one single `proceed` call for the "same:instance" method. As can be seen, each additional active layer adds an approximately constant amount of time to the execution time of a call, and thus the overhead is linear with the number of layers. This result is similar to the performance of EventCJ [21] and ContextJ [6].

Finally, we show the execution time of global activation. As explained above, global activation manipulates all instances of classes that have layers controlled

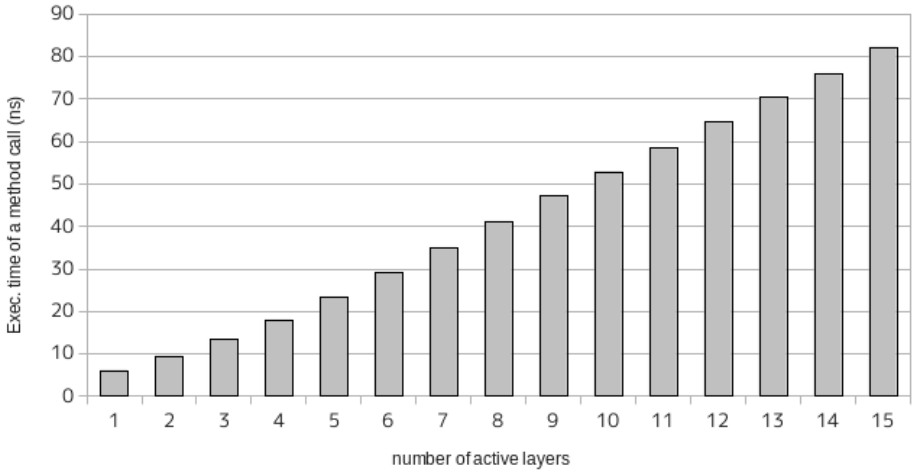


Fig. 17. Execution time of a method call in ServalCJ when increasing the number of active layers. We ran the benchmarks 10 times; the range of error was approximately 0.04% – 0.1%.

by the global context group; thus, the number of such instances affects the execution time of the global activation. The execution time of global activation is measured similarly to the method used in the JGFMethodBench to measure the execution time of a method call. We repeatedly generated an event that activates a layer and an event that deactivates the layer within a loop (we assume that both layer activation and deactivation take the same amount of time). We repeated this experiment while changing the number of target instances.

Fig. 18 shows the results. We can observe that each additional target instance adds a constant amount of time to the execution time of an activation. In the case of a large number of target instances, the layer activation may take more than 1 ms. This overhead will not produce a severe problem if the number of instances with layers is not very large, or if the layer activation does not occur frequently. We consider that most COP applications satisfy these conditions. For example, the environment or a user's current task does not change frequently within a very short period.

8.3 Performance Evaluation with a Maze-Solving Simulator

We evaluated the performance impact of ServalCJ on a real application by estimating the amount of overhead generated in the maze-solving robot simulator. As the microbenchmark results in the previous section show, the amount of overhead depends on several parameters such as the number of active layers. Therefore, we measured those parameters in the application and applied them to the microbenchmark results.

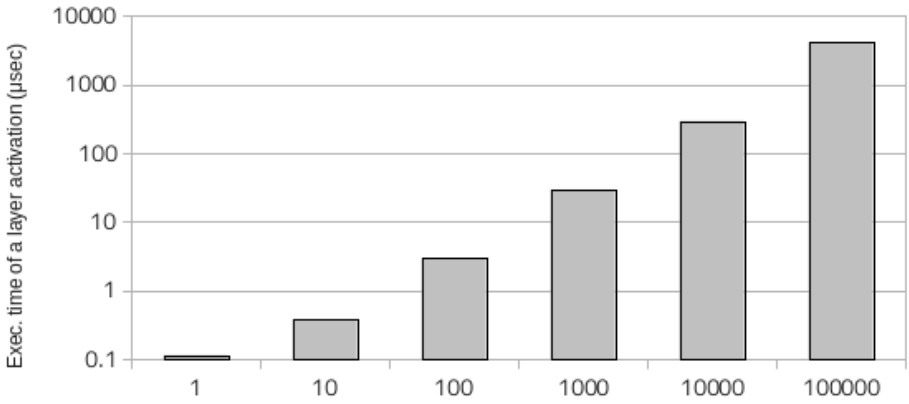


Fig. 18. Execution time of a global activation in ServalCJ when increasing the number of target instances. We ran the benchmarks 10 times; the range of error was approximately 0.01% – 0.08%.

First, we measured the number of active layers, which was at most four. As discussed in Section 6, there are seven layers. Among these seven layers, `EditingMaze`, `SolvingMaze` and `RunningMaze` are exclusive, and among the other four, `RightHandRule` and `Tremaux` are exclusive. Thus, the number of active layers is four when the debugging mode is selected (the `Debugging` layer contains a control-flow that activates `UnderDebugging`).

We then estimated the number of subscribers for global activation. As illustrated in Fig. 9, the context group `MazeUI` is declared as global. This context group has five activate declarations. Among them, `UnderDebugging` changes the behavior of the instances of classes `Edge` (segment) and `Node` (intersection), and the other four activate declarations change the behavior of `Robot` and `View` (Fig. 8). While each instance of the latter two classes are singletons, the number of instances of the former two classes depends on the size of the maze. When we used the maze-solving robot in our classroom, the amount of `Edge` and `Node` instances for the most complicated maze were 43 and 39, respectively. Thus, the total number of subscribers for global activation was 84 (including two singleton instances). According to the microbenchmarks, the overhead of each global layer (de)activation in this case should be less than 3.0 μsec.

To determine the actual overhead of global activation in the maze-solving simulator, we measured the total execution time of global activation using a profiler. We conducted this experiment because, even though we believe that layer (de)activation does not occur very often, the simulator example provides a worse case whereby the `UnderDebugging` layer is activated periodically within the loop statement when solving the maze. Our experiment (de)activated the five layers in Fig. 9. We consider that the overhead was dominated by the cost

of (de)activation of `UnderDebugging`, which is (de)activated. periodically[14] To measure the worst case, profiling was performed in a setting wherein the `Debugging` layer was always active, implying that activation and deactivation of `UnderDebugging` always occurred when refreshing the display. Since each layer activation code was compiled into an advice of AspectJ, we measured the execution time of each method compiled from those advices. We used the profiler included in the Oracle NetBeans IDE 8.01. The total execution time of global layer activation was approximately 25.7 ms, while that of the application was 5,870 ms (both are CPU times). Thus, the overhead was 0.4%, which should be acceptable in most cases.

9 Related Work

COP Related Mechanisms. ContextJS [24] supports user-definable activation mechanisms using the meta-programming features in JavaScript. This is sufficiently powerful to realize any type of layer activation. However, it is nearly impossible to reason about it mechanically because that constitutes meta-programming. Due to its ability to change behavior dynamically, context-aware applications are occasionally error-prone, and providing control of layer activation to the programmer may easily lead to poor application design. Thus, it is preferable to support a more disciplined layer activation mechanism implemented in the programming language.

There are several linguistic mechanisms similar to conditionals in ServalCJ. In LEAD/LEAD++ [1, 2], a method consists of a number of implementations with a condition, and only the implementation where this condition holds is selected for execution. The condition changes with respect to the states of the so-called *metaobjects*, and the programmer can change these states. Tanter et al. proposed context-aware aspects [30], i.e., aspects whose behaviors depend on contexts. This concept is realized as a framework where a context is defined as a pointcut. This is similar to AspectJ's `if` pointcut, but it can also restrict the *past* contexts. Contexts are composable, because they are realized as pointcuts.

Context traits [16] mix the mechanism of trait composition with COP. Context traits take a different approach from that of layer-based COP in that the order of layers is resolved by the programmer. They provide primitive layer activation mechanisms; however, only global activation is supported.

Related Mechanisms Beyond COP. There are also language mechanisms beyond COP, such as aspect-oriented programming (AOP) and event-based programming mechanisms. Generally, there are two major differences between them: (1) while COP emphasizes on changing the behavior of multiple modules *simultaneously*, many of the other mechanisms are essentially intended to change the behavior of each module, and (2) while COP separates context changes from the execution of behavior that depends on contexts, the other mechanisms focus

[14] By "overhead," we mean the overhead against the mechanism where the global activation time is constant with respect to the number of instances.

on control of the execution points where such behavior is executed. We further discuss the similarities and differences between our approach and each of the related mechanisms in the following.

A ServalCJ's event is equivalent to a join-point in AOP. In this sense, ServalCJ's layer activation mechanism is similar to typical AspectJ pointcuts [23] because it provides declarative events using a pointcut language. However, ServalCJ's events can also be conditional. Although layer activation using a conditional expression can be encoded in an AspectJ pointcut, e.g., "`call(* *.*(..)) && if(..)`," this may lead to serious performance degradation.

EventJava [13] is an extension of Java that integrates events with methods. In EventJava, events are broadcast as in the case of global layer activation in ServalCJ. Dynamic subscription of event receivers in event-based languages was proposed in Ptolemy [28]. ServalCJ integrates such event-based mechanisms with dynamic activation of layers in COP. However, a more complex event composition mechanism [25, 26] is currently not supported by the event model in ServalCJ.

Method slots [33] unify event-based programming and AOP by extending the "slots" in Self [31] to hold multiple function closures for each method slot. We can add function closures to each method slot dynamically. Unlike COP mechanisms, this addition is performed in a per-method manner.

To represent context-dependent behavior, other approaches can be taken by representing contexts as objects that are explicitly (or indirectly through dependency injections like Scala's cake pattern [27]) passed to a method. Even though the obliviousness of the layer activation in our approach may make it difficult to predict the base program behavior, it has its own advantage in that it can modularize dynamic behavior changes. The reasoning about properties of context-dependent behaviors described in the discussion part of Section 6 may alleviate this disadvantage.

10 Conclusions

This paper has summarized the differences and commonalities of existing COP languages and proposed a unified model of COP mechanisms and a new COP language, ServalCJ, based on the proposed model. The model represents contexts that specify the duration of layer activation and a set of subscribers that specifies which targets the activation affects. The order of active layers is defined such that synchronous layer activation always has higher priority than asynchronous layer activation. ServalCJ implements this model by providing context groups that can be used to define layer activation based on contexts. ServalCJ covers all the use cases that can be implemented by existing COP mechanisms as well as some other cases that existing COP mechanisms cannot address. The feasibility of the proposed approach has been validated through implementation of a ServalCJ compiler and a performance evaluation.

References

1. Noriki Amano and Takuo Watanabe. LEAD: a linguistic approach to dynamic adaptability for practical applications. In *Proceedings of the IFIP TC2 WG2.4 Working Conference on Systems implementation 2000 : languages, methods and tools*, pages 277–290, 1998.
2. Noriki Amano and Takuo Watanabe. LEAD++: an object-oriented language based on a reflective model for dynamic software adaptation. In *Technology of Object-Oriented Languages and Systems (TOOLS 31)*, pages 41–50, 1999.
3. Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. In *COP'11*, 2011.
4. Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Context holders: Realizing multiple layer activation mechanisms in a single context-oriented language. In *FOAL'14*, pages 3–6, 2014.
5. Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *COP'09*, pages 1–6, 2009.
6. Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, 2011.
7. Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the development of context-dependent Java application with ContextJ. In *COP'09*, 2009.
8. 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.
9. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD'05*, pages 87–98, 2005.
10. 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.
11. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java Grande Applications. In *Proceedings of ACM 1999 Java Grande Conference*, pages 81–88, 1999.
12. 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.
13. Patrick Eugster and K.R. Jayaran. EventJava: An extension of Java for event correlation. In *ECOOP'09*, volume 5653 of *LNCS*, pages 570–594, 2009.
14. Sebastián González, Micolás 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.
15. 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.

16. Sebastián González, Kim Mens, Marious Colacioiu, and Walter Cazzola. Context traits: Dynamic behaviour adaptation through run-time trait recomposition. In *AOSD'13*, pages 209–220, 2013.
17. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
18. Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *FOAL'11*, pages 19–23, 2011.
19. Atsushi Igarashi, Robert Hirschfeld, and Hidehiko Masuhara. A type system for dynamic layer composition. In *FOOL'12*, 2012.
20. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
21. 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.
22. Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Introducing composite layers in EventCJ. *IPSJ Transactions on Programming*, 6(1):1–8, 2013.
23. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisword. An overview of AspectJ. In *ECOOP'01*, pages 327–353, 2001.
24. Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming*, 76(12):1194–1209, 2011.
25. Somayeh Malakuti and Mehmet Aksit. Event modules: modularizing domain-specific crosscutting RV concerns. *TAOSD*.
26. Somayeh Malakuti and Mehmet Aksit. Evolution of composition filters to event composition. In *SAC'12*, pages 1850–1857, 2012.
27. Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA'05*, pages 41–57, 2005.
28. Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*, pages 155–179, 2008.
29. Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *AOSD'12*, 2012.
30. Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *SC 2006*, volume 4089 of *LNCS*, pages 227–242, 2006.
31. David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA'87*, pages 227–241, 1987.
32. 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.
33. Yung Yu Zhuang and Shigeru Chiba. Method slots: Supporting methods, events, and advices by a single language construct. In *AOSD'13*, pages 197–208, 2013.