Generalized Layer Activation Mechanism through Contexts and Subscribers

Tetsuo Kamina
Ritsumeikan University, Japan
kamina@acm.org

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

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

Abstract

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, because these mechanisms interfere with each other in terms of extent (time duration) and scope (a set of units of computations) of activation, it is not trivial to combine them into a single language. 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 COP language called ServalCJ. Contexts specify the extent of activation through temporal logic terms, and subscribers specify the scope of activation through operators provided by the language. We implement a compiler of ServalCJ, and demonstrate its expressiveness by writing a couple of application programs.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Context-oriented programming; Language design and implementation; ServalCJ

1. Introduction

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

MODULARITY'15, March 16–19, 2015, Fort Collins, CO, USA.

http://dx.doi.org/10.1145/2724525.2724570

Context-oriented programming (COP) [16] 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 thus far, and they have successfully modularized such context-dependent behavior [5, 7, 11, 13, 18, 22, 27, 30].

However, existing COP languages have different layer activation mechanisms, making them rather use-case-specific languages. These layer activation mechanisms were 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 from them based on use cases. Furthermore, existing layer activation mechanisms are hard-wired into the language and hence do not provide any means to extend themselves when they are combined with other mechanisms provided by other languages. For example, the per-control-flow activation in ContextJ [5] and JCop [7] is strongly coupled with the current execution thread. Similarly, the implicit activation mechanism in PyContext [30] cannot represent per-instance layer activation.

This issue is exacerbated by the fact that different use cases may coexist in the same application. Thus, there is a natural requirement to generalize existing layer activation mechanisms into one single mechanism.

This paper aims to propose a generalized model of layer activation mechanisms that covers all the existing COP languages, and to develop a COP language based on that model.[1] To do this, we need to solve two problems. First, we need to provide a general model to specify a context and the units of computation to which it is applied. In general, 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 need to unify existing COP mechanisms, which may interfere with each other. Thus, we need to resolve such interference in order to meet the programmer's 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 the existing layer activation mechanisms can be uniformly explained using a single model. Furthermore, we define the dynamic semantics of

[1] This paper is an extended version of our previous work [20] that only presented an idea of the language design. This paper proposes a model of a generalized activation mechanism, simplifies the language design, presents its implementation, and evaluates its performance.

layer activation in the model that meets the programmer's expectations when different existing activation mechanisms coexist in the same application. In this model, the interferences between existing COP mechanisms are resolved by unifying per-instance and global activations, as well as by deciding the order of active layers that are activated synchronously as well as asynchronously.

Based on this model, we designed a language called ServalCJ. A context in ServalCJ is defined as a term of simple temporal logic with a call stack, which can represent the extent of layer activation specified by all the existing layer activation mechanisms (as far as we know). Each context can also be parametrized, which enables 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 they are created). A *context group* in ServalCJ specifies a combination of contexts and subscribers.

The effectiveness of ServalCJ is demonstrated by writing a couple of application programs. The first example is a context-aware program editor, where each construct in ServalCJ is explained. We also present the case study of a maze-solving robot simulator to study the usefulness of ServalCJ. This simulator has different layer activation scenarios; some of them are supported by existing languages, but others are not. We show that these scenarios are uniformly represented by ServalCJ.

To study ServalCJ's feasibility, we implemented a ServalCJ compiler. This compiler translates ServalCJ programs into standard Java bytecode, and thus, they can be run on standard Java virtual machines. We evaluated the performance of method dispatch 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 significant overhead on the running application.

The rest 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 of achieving it. In Section 4, we present a model of a unified activation mechanism, and discuss the appropriate dynamic semantics of the 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 discuss the implementation of the ServalCJ compiler and evaluate its performance. Section 8 discusses related work and, finally, Section 9 concludes this paper.

2. Existing COP Mechanisms

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

2.1 Example

CJEdit, first implemented by Appeltauer [6], 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 a rich text format (RTF) 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 editing. For example, when the programmer is editing code, CJEdit displays an outline view of the program so that he/she can easily determine the structure of the program; when the programmer is editing comments, it displays tools and menus for changing text fonts, sizes, and so on.

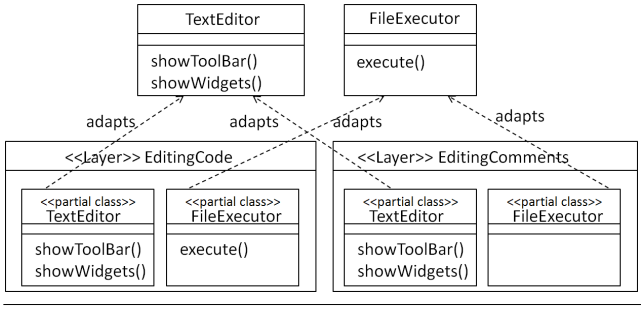


Figure 1. Relationship between layers and classes

In this paper, we extend this program editor with a number of features. First, the program editor is multi-tabbed so that the programmer can open a number of files simultaneously. 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. When the editor is being used online, the files are stored on a repository over the network; when no networks are available, an icon is displayed indicating that the system is operating offline, and then the files are stored on the local disk. Furthermore, we added the find-name function to this program editor; we may search for the names of variables, methods, and classes throughout the entire source code. During the search, the the mouse cursor icon is changed 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 availability of the networks. In the following sections, we refer to such situations as contexts. A COP language provides a modularization mechanism for implementing related context-dependent behavior into one single layer and a layer activation mechanism for dynamically composing and decomposing layers with the application.

2.2.1 Layers

Figure 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 variations of behavior that are executable only when the cursor is on code or comments, respectively. A layer in COP contains a set of partial methods. In Figure 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 (Figure 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 a lower priority is executed.

2.2.2 Layer Activation

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

2.3 Different Mechanisms for Layer Activation

Whereas, for layers, most COP languages provide similar mechanisms, for layer activation, existing COP languages provide a variety of different mechanisms. Each mechanism 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 [5, 7, 11]. 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 it.

Another feature of the per-control-flow activation is that, in this model, a programmer likely to be 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, the ones defined in RenderingCode have priority: 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 [13, 14]. For example, in Subjective-C [13], the activation and deactivation of a layer is written as

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

EventCJ [18] 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.

The layer switching upon event is declaratively specified 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 of per-instance activation in EventCJ is that it can specify only instances that are accessible from the join-point where the event is generated. If these instances cannot directly be obtained from the join-point, we either need to specify a complex chain of method calls or we need to 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 an activation are implicitly specified by a condition. This mechanism is supported by PyContext [30], 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:

```
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 [4]. 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 the per-control-flow activation.

3. Problem Statements

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

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 of 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 that are triggered, for example, by a change in the position of the cursor. On the other hand, if we choose the event-based model, it becomes 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 availability of the network are applied to the entire application, while the status of the opened file may 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 contents displayed on the timeline (all tweets from all followed accounts, tweets only from a specific account, or all tweets that mach a search keyword). The trigger of a context change can be not only an event, such as clicking on a tab, but also implicitly defined through the content of the timeline. The effect of behavior changes may also vary. Each tab can dynamically change its behavior, and its effect is restricted only to the instances contained within the tab. We may also consider other cases, such as behavior changes with respect to the battery status that may affect the entire application. Another example is a pedestrian navigation system that changes its behavior with respect to changes in situation, such as from an indoor to an outdoor environment, which is triggered by an event. In addition, it can change is 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 [22] supports global activation as well as per-control-flow activation as pre-defined activation mechanisms. Although these languages enable us to uniformly represent different cases of behavioral changes to some extent, the activations that they support are still limited. For example, neither language supports implicit activations.

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

3.2.1 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 of the executing machine is running out. Suppose also that activation on some instances is controlled in a per-instance manner to enable the user to control the behavioral changes on these instances manually. For example, the user may require some objects to produce precise computation results during just a few short periods of time even when the battery is on the verge of running out.

In fact, EventCJ does not support such a situation, because a global activation always cancels a per-instance deactivation. In EventCJ, the active 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 would be likely to arise if the different linguistic mechanisms were "piled up" into one single language.

3.2.2 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: the inner-most layer always precedes the others. 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 conflicts with that 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`, 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 the layer activation that is asynchronously triggered by events will occur.

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 uniformly define the activation order.

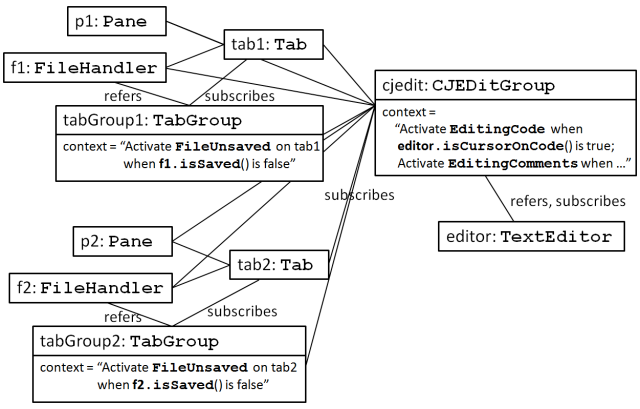


Figure 2. Unified model of existing COP mechanisms

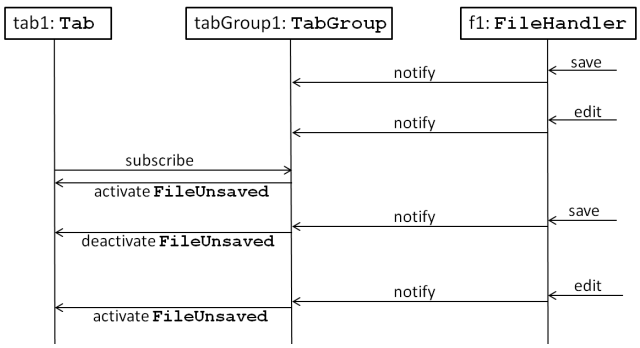


Figure 3. Dynamic subscription and layer activation

4.1 Contexts and Subscribers

To develop the generalized activation model, we coordinate the different layer activation mechanisms in the existing COP languages using the following two concepts: *context*, which specifies the time and duration of the layer activation 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 a context group, all the objects subscribing to that group will start searching partial methods in that layer upon method dispatch. For example, the contexts specifying from when to when the cursor is on code and comments, respectively, affect the entire application with respect to the behavior of the toolbar and menubar, and thus they are grouped into one single context group. The context specifying when the file opened in a tab is unsaved affects only a limited subset of instances in the application, and thus they are grouped into another context group.

We illustrate this model in Figure 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 parametrized; in `cjedit`, this parameter is bound to `editor`, an instance of `TextEditor`. When the state of `editor` changes, the layer activation on all the subscribed instances also changes. Similarly, the instance `tabGroup1` of the `TabGroup` context group specifies the contexts for activating `FileUnsaved`, which implements the behavior relating to unsaved files. Only an instance `tab1` of `Tab` subscribes to that context group. The context specified in `TabGroup` is also parametrized, and this parameter is bound to `f1`, an instance of `FileHandler`.

We further illustrate the dynamic semantics of this model using the UML sequence diagram in Figure 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 `tagGroup1`, these notifications do not trigger any layer activation. After an instance of `Tab`, namely `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`.

Table 1. Existing COP languages categorized in our model

	Global	Thread	Instance
Control-flow		ContextJ, PyContext	Context-Erlang[27]
Imperative	Subjective-C		ContextErlang
Event-based	EventCJ		EventCJ
Implicit	Flute[9]	PyContext	

We show that each existing COP language falls into one specific case of this model, as illustrated in Table 1. In this table, the methods that specify contexts are categorized into four variants, per-control-flow, imperative, event-based, and implicit, corresponding to each layer activation model discussed in Section 2.3. In this table, the methods that specify subscribers are also categorized into three variants: global (specifying all instances in the application), thread (specifying the currently executing thread), and instance (specifying 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 9. For example, EventCJ supports event-based specifications of contexts that are applicable both to 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, the implicit activation for a limited set of instances is not supported by any existing COP languages.

4.2 Model of Activation Order

The synchronous-asynchronous interference explained in Section 3.2 implies that we need to separately manage the synchronous and asynchronous layer activation. To meet the programmer's expectation, in our model the synchronous layer activation always precedes the asynchronous one. More precisely, the semantics of layer activation in our model is 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, the global and the 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.[2] Let $\bar{\mathtt{L}}_S = \mathtt{L}_1, \cdots, \mathtt{L}_n$ be a sequence of layers that are synchronously activated, and let $\bar{\mathtt{L}}_A = \mathtt{L}'_1, \cdots, \mathtt{L}'_n$ be a sequence of layers that are asynchronously activated. 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 L and returns a new concatenation of 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 the synchronous layer activation. If L is not contained in both $\bar{\mathtt{L}}_A$ and $\bar{\mathtt{L}}_S$, it is added at the head of the sequence $\bar{\mathtt{L}}_S$, indicating that L has the highest priority. Otherwise, L is removed from the original position and moved to the head of the sequence $\bar{\mathtt{L}}_S$.

Similarly, the 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 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 L has a higher priority than all layers in $\bar{\mathtt{L}}_A$, but a lower 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 active layers does not change, because this case indicates that L already been active with a higher priority than the layers in $\bar{\mathtt{L}}_A$.

We define the function *deact* to model the 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 Appendix A. For example, *actSync* is always used when the `with`-block is applied, and *actAsync* is always used when the 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 the partial method starts from the right-most layer of $\bar{\mathtt{L}}_S$ and proceeds to to the left-most layer of $\bar{\mathtt{L}}_A$. If no partial methods are found, the original method is dispatched.

To address the 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 the instances that have that layer. This mechanism ensures that a global activation does not interfere with per-instance ones, but at the cost of activating the layer for all these instances.

For example, in the piece of code in Section 3.2, the layers activated by `with`-blocks 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 becomes

```
EditingComments;EditingCode,RenderingCode
```

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

5. COP Language with Contexts and Subscribers

We designed the COP language ServalCJ to be an instance of the generalized activation model discussed in Section 4. It 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.

[2] As illustrated in Section 3.2.2, 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 reserved for future work. The operational semantics discussed in Appendix A does not change if this ordering is changed.

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

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

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 [4], 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 outside the scope of this paper.

We formalize the dynamic semantics of ServalCJ in Appendix A. 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 instance of context group contains subscribers, that is, a 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.

Figure 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, 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, as specified by line 2 of Figure 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 Figure 4 declares when the layer `FileUnsaved` is active, which occurs whenever the `isSaved` method call on `f` results in `false`. We discuss the specification of layer activation further in Section 5.2.

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

Figure 5. Example of a global context group

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, called a *global context group*.

Figure 5 shows an example of a global context group declaration. To make the context group global, we need to provide the modifier `global`. A global context 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; it becomes effective only after the instance creation. The context group `CJEditGroup` in Figure 5 declares two layer activation rules: (1) the layer `EditingCode` is active whenever the `isCursorOnCode` method call on `editor` results in `true` and (2) the layer `EditingComments` is active whenever the `isCursorOnComments` method call on `editor` results in `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, meaning that, when this term is `true`, the layer is active. This specification is performed using an *activate declaration*, which has the syntax

`activate` *LayerName Context* ;

This declaration starts with the keyword `activate` followed by the name of the layer. We next specify a context, which has a `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 their names, and *composite contexts* that are contexts combined by using logical-OR, logical-AND, and NOT expressions. We further discuss each of these in the following sections.

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

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

Within the `if` expressions, we can use any Boolean-type Java expressions. We note that ServalCJ can represent implicit activation that is applied per-instance. In Figure 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 referred to 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 fired like a method invocation. For example, in Figure 6, two events, `MoveOnCode` and `MoveOnComments`, are declared in the class `TextEditor`. These events are fired during the execution of `onCursorPositionChanged` and if the `isCursorOnCode` (`isCursorOnComments`, resp.) call results in `true`. We can also declare an event using the AspectJ pointcut language.

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

Figure 6. Publishing events in ServalCJ

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

```
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 becomes active whenever the event `MoveOnCode` is fired and becomes inactive whenever the event `MoveOnComments` is fired.

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` becomes active when `MoveOnCode` is fired and inactive when `MoveOnComments` is fired *only when these events are fired by* `editor`. It should be noted that we cannot specify an event sender in EventCJ.

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

```
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. It should be noted 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 as specified in the cflow expression.

Per-thread activation. The `with`-block-based COP languages, such as ContextJ, activate layers in a per-thread manner. We note that most useful cases of ContextJ are easily encoded by the combination of a global context group and the cflow activation introduced in Section 5.2. To restrict the effect of layer activation to the currently executing thread, we may introduce another modifier,

perthread, that states that the set of subscribers consists of all subscribers *when they are accessed from the thread executing the control flow*:

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

The perthread modifier does not produce any effects when it is used with other expressions.

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

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

This declaration starts 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 it should be enclosed within the 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 declaration

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

ServalCJ also provides a way to compose contexts to represent a more complex layer activation. This composition was originally known as *composite layers* [19]. To compose contexts, we can use the logical operators || (logical-OR), && (logical-AND), and ! (NOT).

6. Case Study

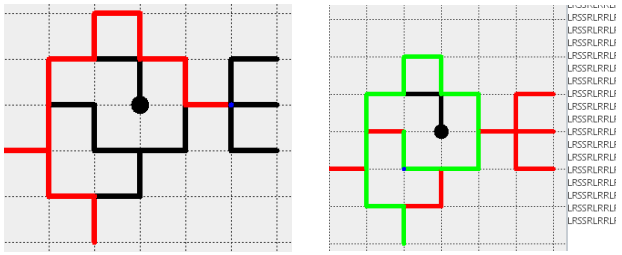
The program editor example described above shows how different COP mechanisms coexist in the same application, justifying our design of a generalized layer activation mechanism in ServalCJ.

To provide more evidence illustrating such situation, we conducted another case study to implement a maze-solving simulator.[3] This application simulates how a line-tracing robot solves a maze. The following code skeleton illustrates how the robot performs maze-solving.[4]

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

[3] The source code of this simulator is available at https://github.com/ServalCJ/mazesimulator.git.

[4] This case study is 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. The red lines represent the traced path.

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

Figure 7. The maze-solving simulator. The lines indicate paths within the maze. The black circle represents the goal.

The followSegment method performs line-tracing until the robot reaches an intersection, a corner, or a dead-end (in the following, we call them *intersections* for simplicity). 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 the (possibly) optimized path from the starting point to the current intersection by eliminating the dead-ends. The robot repeats these behaviors until it reaches the goal. After solving the maze, the robot can run the optimized path from the starting point to the goal by simply following the path calculated by simplify.

If the maze contains loops, the robot also needs to remember all the visited intersections and/or segments (by a segment, we mean 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, while others can solve more general 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 its coordinates to indicate its position. The instance robot of Robot emulates maze-solving on 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: the left-hand rule, right-hand rule, and Trémaux's algorithm.[5] The selection of these algorithms changes the behavior of turn and possibly that of simplify.

For the user, this simulator provides a number of functionalities: editing a maze, simulating how the robot solves the maze, and simulating how the robot follows the optimized path after solving the maze. These functionalities 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 functionalities 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 automatically switched when the functionalities are switched. During the maze-solving, the visited intersections and segments are colored to visualize the traced path (Figure 7(a)). Furthermore, while the robot is solving the maze, the user can select a debug mode that displays

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

```
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() { .. }
  }
}
```

Figure 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
6      from solved to neverMatchingEvent;
7    activate Debugging
8      from startDebug to endDebug;
9    context Print is
10     in cflow(call(void Simulator.print()));
11   activate UnderDebugging
12     when Debugging && when Print;
13 }
```

Figure 9. Context group for globally activating layers

```
1  contextgroup Algorithm(Robot robot) {
2    activate RightHandRule
3      if(robot.isRightHandRule());
4    activate Tremaux if(robot.isTremaux());
5  }
```

Figure 10. Context group applicable to the robot instance

the currently calculated optimized path by printing the text representing the optimized path and changing the color of intersections and segments in the optimized path (Figure 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 for solving 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 the textbox where the text representing the currently calculated optimized path is printed.
- `UnderDebugging` changes the color of segments and intersections in the maze only if they are in the optimized path and the debug mode is selected.

Note that the debugging feature is divided into two layers, `Debugging` and `UnderDebugging`, because, as explained below, they are applied in slightly different situations.

These layers change the behavior of multiple classes. For example, `SolvingMaze` and `RunningMaze` change the appearance of 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 (Figure 8).

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

Figure 9 shows the context group for managing globally activated layers. It specifies activate declarations for five layers. The activations for the former four layers are controlled by the from-to expressions. The events that activate and deactivate 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. The `UnderDebugging` layer changes how the color of visited segments and intersections is set:

```
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;
    }
  }
}
```

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 under the control flow where the shortest path is printed (which also calls the `setTraced` methods on `Edge` and `Node`). We apply the `cflow` expression in this case.

Figure 10 shows the context group for managing activations that are applicable to a specific robot instance. Although there exists only one single robot instance in this application, we apply the per-instance activation in this case for future extensibility (e.g., supporting multiple robots that execute different algorithms). In this case, we apply the conditional (`if`) expressions instead of the from-to expressions to specify activate declarations. We do this because, in the base program, the value indicating the algorithm is

set to the robot instance when the user selects the algorithm, which is useful for judging which layer should be activated. We note that the program structure of the base program may affect the decision taken by the programmer to select the activation mechanism.

Discussion We first discuss the appropriateness of applying COP to implement this simulator.[6] First, the variations of context-dependent behavior mentioned in this simulator crosscut multiple classes and are modularized by corresponding layers in COP. For example, layers `SolvingMaze` and `RunningMaze` change the behavior in both the simulator and GUI components. `Debugging` also 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 may apply meta-programming techniques to implement dynamic changes of behavioral variations. In such techniques, however, 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 the debug mode. It is difficult for meta-programming to mechanically check such properties. By using COP, on the other hand, we may easily generate a state transition model from the event-driven layer switching to perform model checking [18]. The exclusiveness of algorithms can be checked by only checking the exclusiveness of the simulator states that affect the value of the expressions used in the `if` expressions (such as the value of the `isRightHandRule` call). The dependency between `UnderDebugging` and `Debugging` is immediately obtained from the context specification of the activate declaration.

Finally, COP supports modularization of the specification that determines when the behavior changes occur. If we apply other approaches to implement such behavior changes (such as the state design pattern), the behavior changes may be hard-wired and scattered into the base program. Using the declarative specification of layer switching in COP languages such as JCop [7] and EventCJ [18], such behavior changes are separately specified. Although the examples shown in this paper are written by using the imperative events for brevity, ServalCJ also supports declarative events using AspectJ-like pointcut language.

We further compare ServalCJ with existing COP languages. The case study showed that different activation mechanisms are used in the same application. As discussed in Section 2.3, no existing COP languages can support such a variety of activation mechanisms. There are no existing COP languages that support all the event-based, per-control-flow, and implicit activation mechanisms, while ServalCJ supports all of them (the imperative activation in Subjective-C can also be represented by the from-to expression where the until clause specifies an event that will never happen). Furthermore, the case study shows how several *combinations* between activation mechanisms and sets of subscribers can be used in the same application. In particular, the combination of global and per-control-flow activation as well as that of per-instance and implicit activation are used in the application. As summarized in Table 1, existing COP languages do not solely support these 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 dynamically subscribe to the context group, and we can specify the sender of the event.

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

7. Implementation

The compiler of ServalCJ is built on top of the AspectBench Compiler (abc) [8] by extending the front-end. The compiler eventually generates bytecode executable on the standard Java virtual machine by first translating a ServalCJ program into an AspectJ program, and then by letting the AspectJ compiler generate bytecode.[7]

7.1 Overview of Translation

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

We first explain how layers are translated. The translation is similar to what performed in the EventCJ compiler [18]. 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 the code that first obtains the list of instances of active layers (i.e., instances of those inner classes) and then calls the instance method at the tail position of the list. The proceed call is translated to the code that calls the method on the instance at the preceding position of the list. Every class that is extended by partial methods will have a new field, `lm`, in order to store a list of active layers.

The conditional expressions are evaluated just before the call of a partial method. Precisely, 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, and for each context group, evaluates a conditional expression (associated with that context group). If any of the conditions hold and if the corresponding layer is not active, the code activates the layer. On the other hand, if they do not hold and if the corresponding layer is active, the code deactivates it.

ServalCJ implements global layer activation using the per-instance layer activation mechanism. It places globally active layers in the list of active layers in every instance. To do so, the runtime manages a list of all instances in a program.[8] 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 (which corresponds to the global active layers $\overline{\mathtt{L}}$ that appear in the reduction relation in Appendix A). 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

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

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

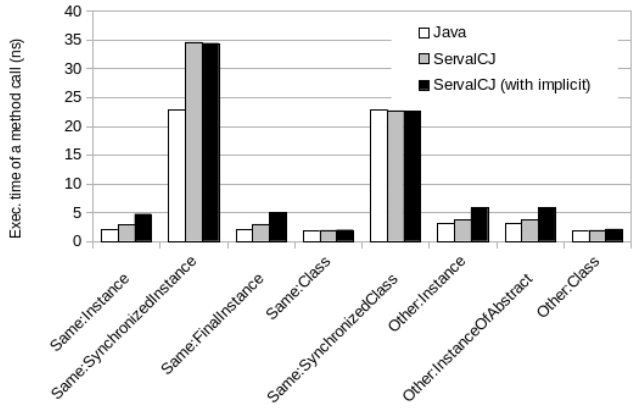


Figure 11. Execution time of a method call in ServalCJ and Java (shorter is better). We ran the benchmarks 10 times; the range of error was approximately 0.007% – 1%.

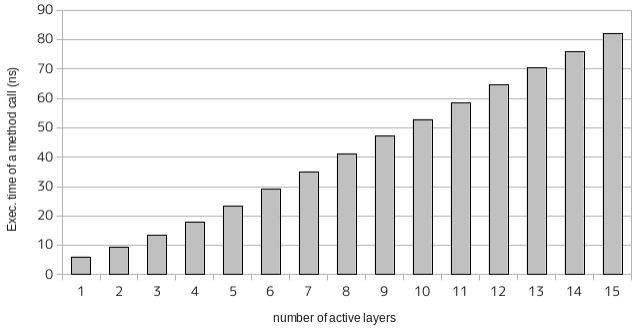


Figure 12. Execution time of a method call in ServalCJ when increasing the number of active layers. We ran the benchmarks 10 times; the range of error was approximately 0.04% – 0.1%.

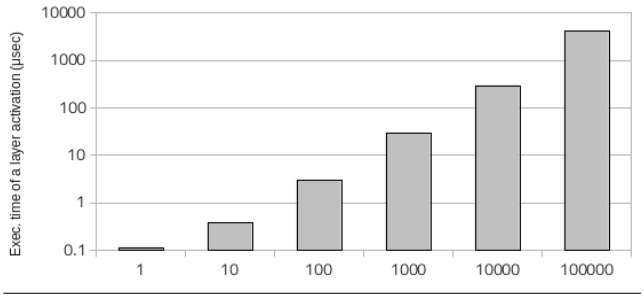


Figure 13. Execution time of a global activation in ServalCJ when increasing the number of target instances. We ran the benchmarks 10 times; the range of error was approximately 0.01% – 0.08%.

updating 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 appropriately handle the recursive calls. For the layer activations that are composed with multiple contexts by the `&&` and `||` operators, the compiler resolves which layers should be active on each join-point specified by the pointcuts.

7.2 Microbenchmarks

In this section, we evaluate the performance of method dispatching 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 that is always imposed on the compiled program, we conducted a number of experiments. The first experiment was performed in order to verify that ServalCJ does not seriously degrade the execution performance when we do not use the ServalCJ specific features that impose additional overheads. The objective of the second experiment was to measure the overhead of layered method calls with implicit activation and global activation.

For the benchmark, we used JGFMethodBench in the Java Grande Forum Benchmark Suite [10] version 2. We extended this benchmark in order 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 on 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. It should be noted that, to prohibit the JIT compiler from eliminating the entire target method call (this elimination prohibits us from measuring the overhead with respect to method calls), we used the client VM instead of the server VM in the benchmark.

Figure 11 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 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 the measurement. We did not provide any partial methods for the class methods, because currently ServalCJ does not support this.

The figure shows that, when no layers are active, the performance of method calls in ServalCJ is comparable with that in plain Java if implicit activation is not used. The primary reason for this is that in this case, ServalCJ does not impose any overhead on the program except that which is incurred when it checks the number of currently active layers. The method call is approximately two times slower if we use implicit activation; this overhead is also comparable with other COP languages.

Figure 12 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. We can see that each additional active layer adds roughly a constant amount of time to the execution time of a call. This result is similar to the performances of EventCJ [18] and ContextJ [5].

Finally, we show the execution time of global activation. As explained above, the global activation manipulates all instances of classes that have layers controlled by the global context group, which means that the number of such instances affects the execution time of the global activation. The execution time of global activation is measured in a way similar to that 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 time). We repeated this experiment while changing the number of target instances.

Figure 13 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 meet these conditions; for example, the environment or a user's current task does not change frequently within a very short period of time.

7.3 Performance Evaluation with a Maze-Solving Simulator

We evaluated 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. We therefore measured those parameters in the application and applied them to the microbenchmark results.

First, we measured the number of active layers, which turned out to be at most four. As illustrated in Section 6, there are seven layers. Among them, `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 Figure 9, the context group `MazeUI` is declared as global. This context group has five activate declarations. Among them, `UnderDebugging` changes the behavior of instances of classes `Edge` (segment) and `Node` (intersection), and the other four activate declarations change the behavior of `Robot` and `View` (Figure 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 number of `Edge` and `Node` instances for the most complicated maze were 43 and 39, respectively. Thus, the total number of the 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 where the `UnderDebugging` layer is periodically activated within the loop statement when solving the maze. Our experiment (de)activated five layers in Figure 9; we consider that the overhead is dominated by the cost of (de)activation of `UnderDebugging`, which is periodically (de)activated.[9] To measure the worst case, this profiling was performed in a setting where the `Debugging` layer was always active, implying that the 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 Oracle NetBeans IDE 8.01. The total execution time of global layer activation was approximately 25.7 ms, while the total execution time of the application was 5,870 ms (both are CPU time). The overhead was thus 0.4%, which should be acceptable in most cases.

8. Related Work

COP related mechanisms. ContextJS [22] supports user-definable activation mechanisms by using the meta-programming features in JavaScript. This is sufficiently powerful to realize any style of layer activation. However, it is almost impossible to reason about it mechanically, as that constitutes meta-programming. Because of its ability to dynamically change behavior, 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 a couple of 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 is changed with respect to the states of the so-called *metaobjects*, and the programmer can change these states. Tanter et al. proposed context-aware aspects [28], that is, 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 is also able to restrict the *past* contexts. Contexts are composable, because they are realized as pointcuts.

Context traits [15] 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, and 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. In general, there are two major differences between them: (1) while COP puts emphasis on changing the behavior of multiple modules *at once*, many of the other mechanisms are basically intended to change behavior of each module and (2) while COP separates context changes from the execution of behavior that depends on contexts, the other mechanisms focus on the control of the execution points where such behavior is executed. We further discuss similarities and differences between our approach and each of the related mechanisms as follows.

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 [21], as 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 like "`call(* *.*(..)) && if(..)`," it may lead to significant performance degradation.

EventJava [12] 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 [26]. ServalCJ integrates such event-based mechanisms with dynamic activation of layers in COP. However, a more complex mechanism of event composition such as that proposed in [23, 24] is currently not supported by the event model in ServalCJ.

Method slots [31] unify event-based programming and AOP by extending the "slots" in Self [29] to hold multiple function closures for each method slot. We can dynamically add function closures to each method slot. 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 [25]) passed on 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 the dynamic changes of behavior. The reasoning about properties among context-dependent behaviors described in the discussion part of Section 6 may alleviate this disadvantage.

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

9. Conclusions

This paper summarized the differences and commonalities of existing COP languages, and proposed a unified model of COP mechanisms and a new COP language, ServalCJ, that is based on this model. The model represents contexts that specify the duration for which the layer activation continues and a set of subscribers that specifies which targets the activation affects. The order of active layers is defined so that synchronous layer activation always has a priority that is higher than that of asynchronous layer activation. ServalCJ implements this model by providing context groups within which we can 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 the existing COP mechanisms do not address. The feasibility of our approach was validated through the implementation of a ServalCJ compiler and its 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] 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.

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

[6] Malte Appeltauer, Robert Hirschfeld, and Hidehiko Masuhara. Improving the development of context-dependent Java application with ContextJ. In *COP'09*, 2009.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *FOAL'11*, pages 19–23, 2011.

[18] 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.

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

[20] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A unified context activation mechanism. In *COP'13*, 2013.

[21] 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.

[22] 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.

[23] Somayeh Malakuti and Mehmet Aksit. Event modules: modularizing domain-specific crosscutting RV concerns. *TAOSD*.

[24] Somayeh Malakuti and Mehmet Aksit. Evolution of composition filters to event composition. In *SAC'12*, pages 1850–1857, 2012.

[25] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA'05*, pages 41–57, 2005.

[26] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*, pages 155–179, 2008.

[27] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *AOSD'12*, 2012.

[28] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *SC 2006*, volume 4089 of *LNCS*, pages 227–242, 2006.

[29] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA'87*, pages 227–241, 1987.

[30] 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.

[31] 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.

A. Formal Model of Layer Activation

We define the formal semantics of layer activation by combining two COP calculi, FECJ [3], which provides per-instance and asynchronous activation, and ContextFJ [17], which provides global and synchronous activation.

A.1 Syntax.

The abstract syntax is shown as follows:

CL	::=	`class C ◁ C { C̄ f̄; K M̄ }`	(*classes*)
K	::=		(*constructors*)
		`C(C̄ f̄){ super(f̄); this.f̄ = f̄; }`	
M	::=	`C m(C̄ x̄){ return e; }`	(*methods*)
e, d	::=	`x` | $\mathtt{e}^{\hat{\ell}}$`.f` | $\mathtt{e}^{\hat{\ell}}$`.m(`$\bar{\mathtt{e}}^{\hat{\ell}}$`)` | `new C(ē)`	(*expressions*)
		| `proceed(ē)` | `with L e`	
		| `v` | `v<C,L̄,L̄>.m(v̄)` | `{e}`	
t	::=	`↑ L` | `↓ L`	(*activation rules*)
ℓ	::=	ι | γ	(*event labels*)
p	::=	`v ↦ new C(v̄)<L̄>`	(*partial stores*)
μ	::=	p̄	(*stores*)
st	::=	ϵ | *st* $\gg$ L̄	(*stack*)

Let 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; 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., $\bar{\mathtt{f}}$ stands for a possibly empty sequence $\mathtt{f}_1, \cdots, \mathtt{f}_n$. We also abbreviate a sequence of pairs by writing "$\bar{\mathtt{C}}$ $\bar{\mathtt{f}}$" for "$\mathtt{C}_1$ $\mathtt{f}_1, \cdots, \mathtt{C}_n$ $\mathtt{f}_n$," where n denotes the length of $\bar{\mathtt{C}}$ and $\bar{\mathtt{f}}$. Similarly, we write "$\bar{\mathtt{C}}$ $\bar{\mathtt{f}}$`;`" as shorthand for the sequence of declarations "$\mathtt{C}_1$ $\mathtt{f}_1$`;`$\ldots$ $\mathtt{C}_n$ $\mathtt{f}_n$`;`" and "`this.`$\bar{\mathtt{f}}$`=`$\bar{\mathtt{f}}$`;`" as shorthand for "`this.`$\mathtt{f}_1$`=`$\mathtt{f}_1$`;`$\ldots$`;this.`$\mathtt{f}_n$`=`$\mathtt{f}_n$`;`". We use commas and semicolons for concatenations. We abbreviate a concatenation $\bar{\mathtt{L}}_A; \bar{\mathtt{L}}_S$ of asynchronously activated layers $\bar{\mathtt{L}}_A$ and synchronously activated layers $\bar{\mathtt{L}}_S$ simply as a sequence of layers $\bar{\mathtt{L}}$ 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. We also use a hat to denote an optional element, i.e., $\hat{\ell}$ denotes that there is either a label ℓ or no labels. An empty element is denoted by ϵ, which is usually omitted.

The runtime expression `{e}` appears only as a subterm of `with` under reduction. The runtime expression `new C(`$\bar{\mathtt{v}}$`)<C,`$\bar{\mathtt{L}}'$`,`$\bar{\mathtt{L}}$`>.m(`$\bar{\mathtt{e}}$`)`, where $\bar{\mathtt{L}}'$ is assumed to be a prefix of $\bar{\mathtt{L}}$, means that `m` is going to be invoked on `new C(`$\bar{\mathtt{v}}$`)`. The annotation `<C,`$\bar{\mathtt{L}}'$`,`$\bar{\mathtt{L}}$`>` indicates the cursor where method lookup should start.

A label attached to an expression denotes an event receiver that simplifies the asynchronous layer activation; i.e., the expression $\mathtt{e}^{\iota}$ represents that an event (that activates some layer) is received by `e`. Similarly, $\mathtt{e}^{\gamma}$ represents that an event, which globally activates some layer, is sent by `e`. A synchronous layer activation is modeled by a `with` expression. Even though this calculus does not support all linguistic features provided by ServalCJ (for example, it cannot represent `with` applied to only specific instances), it explains how the interference problems shown in Section 3.2 are resolved.

A value `v` is a location. A store μ is a sequence of pairs of a location and an object. This pair is of the form `v ↦ new C(`$\bar{\mathtt{v}}$`)<`$\bar{\mathtt{L}}$`>`, which is read "an object `new C(`$\bar{\mathtt{v}}$`)<`$\bar{\mathtt{L}}$`>` is stored at location `v`." A stack *st* remembers a sequence of layers $\bar{\mathtt{L}}$ before the reduction of `with` starts so that the computation can restore that sequence after finishes the reduction of `with`.

Unlike the existing COP languages, the calculus does not provide syntax for layers. Partial methods are registered in a partial method table *PT*, which 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 ↑ `L` (activating `L`) or a deactivation ↓ `L` (deactivating `L`).

Intuitively, activate declarations described in Section 5.2, from-to expressions in particular, correspond to the activation rules in *TT*. Asynchronous layer activation corresponds to the layer activation triggered by either conditionals or from-to expressions. Synchronous layer activation corresponds to the layer activation triggered by `cflow`. A value with label $\mathtt{v}^{\ell}$ corresponds to a subscriber in ServalCJ, and the global label γ illustrates the behavior of the global context group.

A program $(CT, PT, TT, \mathtt{e})$ 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 a well-formed expression `e` that corresponds to the body of the main method. We assume *CT*, *PT*, and *TT* to be fixed and to satisfy the following sanity conditions:

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

A.2 Operational Semantics

The operational semantics is given by a reduction relation of the form $\mathtt{e} \mid \mu \mid \bar{\mathtt{L}} \mid st \longrightarrow \mathtt{e}' \mid \mu' \mid \bar{\mathtt{L}}' \mid st'$, which is read "expression `e` under a store μ, global active layers $\bar{\mathtt{L}}$, and a stack *st* reduces to $\mathtt{e}'$ under μ', $\bar{\mathtt{L}}'$, and *st*′." We assume that neither μ nor μ' contain duplicate names. We only show the rules relevant to discussing the order of active layers. For the other rules, the reader may consult [3].

The following rule represents the reduction that occurs when a value `v` receives an event denoted by label ι.

$$\frac{TT(\iota) = \uparrow \mathtt{L} \qquad \mu(\mathtt{v}) = \mathtt{new\ C(}\bar{\mathtt{v}}\mathtt{)<}\bar{\mathtt{L}}'\mathtt{>} \qquad actAsync(\bar{\mathtt{L}}', \mathtt{L}) = \bar{\mathtt{L}}'' \qquad \mu' = (\mathtt{v} \mapsto \mathtt{new\ C(}\bar{\mathtt{v}}\mathtt{)<}\bar{\mathtt{L}}''\mathtt{>}, \mu)}{\mathtt{v}^{\iota} \mid \mu \mid \bar{\mathtt{L}} \mid st \longrightarrow \mathtt{v} \mid \mu' \mid \bar{\mathtt{L}} \mid st}$$

This rule obtains the corresponding layer activation rule stored in *TT*, and calculates the order of active layers by applying *actAsync*. Note that we only explain the case for layer activation for simplicity. The layer deactivation case is obtained by replacing ↑ with ↓ and *actAsync* with *deact*. The store μ is updated by inserting the address of the instance with new active layers.

Similarly, the following rule represents the reduction that occurs when a value `v` sends a global event denoted by label γ.

$$\frac{TT(\gamma) = \uparrow \mathtt{L} \qquad actAsync(\mu, \mathtt{L}) = \mu' \qquad actAsync(\bar{\mathtt{L}}, \mathtt{L}) = \bar{\mathtt{L}}' \qquad actAsync(st, \mathtt{L}) = st'}{\mathtt{v}^{\gamma} \mid \mu \mid \bar{\mathtt{L}} \mid st \longrightarrow \mathtt{v} \mid \mu' \mid \bar{\mathtt{L}}' \mid st'}$$

In this rule, we overload the definition of *actAsync*; when it is applied to μ, for each sequence of active layers $\bar{\mathtt{L}}_i$ in the range of μ is updated by applying $actAsync(\bar{\mathtt{L}}_i, \mathtt{L})$. We also overload *actAsync* to be applied to a stack *st* in the similar way. The sequence of global active layers $\bar{\mathtt{L}}$ is also updated. Again, we only show the case for layer activation.

The following rule explains the reduction of an instance creation. It should be noted that the global active layers $\bar{\mathtt{L}}_A$, which are asynchronously activated, are prospectively activated for the new instance.

$$\frac{\mathtt{w} \notin dom(\mu)}{\mathtt{new\ C(}\bar{\mathtt{v}}\mathtt{)} \mid \mu \mid \bar{\mathtt{L}}_A; \bar{\mathtt{L}}_S \mid st \longrightarrow \mathtt{w} \mid (\mathtt{w} \mapsto \mathtt{new\ C(}\bar{\mathtt{v}}\mathtt{)<}\bar{\mathtt{L}}_A\mathtt{>}, \mu) \mid \bar{\mathtt{L}}_A; \bar{\mathtt{L}}_S \mid st}$$

The following rules illustrate synchronous layer activation.

$$\frac{\mathit{actSync}(\overline{\mathtt{L}}, \mathtt{L}) = \overline{\mathtt{L}}'}{\mathtt{with\ L\ e} \mid \mu \mid \overline{\mathtt{L}} \mid st \longrightarrow \{\mathtt{e}\} \mid \mu' \mid \overline{\mathtt{L}}' \mid st \gg \overline{\mathtt{L}}}$$

The *actSync* function ensures that layer `L` is active during the evaluation of body `e`, which is reduced to the runtime expression $\{\mathtt{e}\}$. Stack *st* is updated so that it can pop $\overline{\mathtt{L}}$, the globally activated layers before the evaluation of `with`, when the evaluation of the body is finished. The reduction rules for this runtime expression are given as follows:

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

$$\{\mathtt{v}\} \mid \mu \mid \overline{\mathtt{L}} \mid st \gg \overline{\mathtt{L}}' \longrightarrow \mathtt{v} \mid \mu \mid \overline{\mathtt{L}}' \mid st$$

These reduction rules ensure that the synchronous layer activation always precedes the asynchronous ones, and a `with` expression does not affects the order of asynchronously activated layers outside. It should be noted that each instance `new C(`$\overline{\mathtt{v}}$`)<`$\overline{\mathtt{L}}$`>` has only asynchronously activated layers. Thus, in the method lookup we need to include layers that are globally and synchronously activated in the search sequence, as shown in the following reduction rules for method invocation:

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

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

The cursor for the method lookup is set to be a concatenation of asynchronously activated layers per-instance $\overline{\mathtt{L}}''_A$ and globally and synchronously activated layers $\overline{\mathtt{L}}_S$.