Method Safety Mechanism for Asynchronous Layer Deactivation

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

Atsushi Igarashi
Kyoto University, Japan
igarashi@kuis.kyoto-u.ac.jp

ABSTRACT

We propose a context-oriented programming (COP) language that allows layers to define base methods, while layers can be asynchronously activated and deactivated. Base methods in layers greatly enhances modularity because they extend classes' interface without modifying the original class definitions. However, calling such a method defined in a layer is tricky as the layer may not be active when the method is called. We tackle this problem by introducing a method lookup mechanism that uses the lexical scope of a method invocation to COP; i.e., besides currently activated layers, the layer where the method invocation is written, as well as layers on which that layer depends, are considered for method lookup. We implemented this mechanism in ServalCJ, a COP language that supports asynchronous layer activation as well as synchronous layer activation.

Categories and Subject Descriptors

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

General Terms

Languages

Keywords

Layer-introduced base method, ServalCJ

1. INTRODUCTION

Context-Oriented Programming (COP) is an approach to improve modularity of variations of behavior that depend on contexts [12]. COP languages provide linguistic constructs that modularize such variations using *layers*[1], and that dynamically activate/deactivate them according to the executing contexts [8, 12]. These constructs give advantages to COP in modularity and ensuring consistency in dynamic changes using scoping [8] or model checking [16], compared with over existing object-oriented mechanisms and practices like design patterns [5]. A layer defines *partial methods*, which run before, after, or around a call of method with the same signature defined in a class only when the layer is active. In the rest of this paper, we call a method in a layer that overrides other methods as a *partial method*, and a method in a layer that is not a partial method (i.e., that introduces a new signature) as a *base method in a layer*.

Although many of COP languages only support partial methods, base methods in layers are also known to be useful in COP [13]. For example, an adaptive user interface for a text editor may provide different menu items for the contents opened by that editor. The menu items and their associated behaviors are dynamically changed with respect to the currently opened file: if the user is opening a program, the editor provides an "execute" menu item, and the behavior that triggers an execution of that program is associated with that menu item. This menu item and behavior are defined in a layer, namely `Programming`, where the new behavior is implemented by a base method in that layer. Then, this layer can be overridden by another layer, namely `Debugging`, which is activated only when the user performs debugging. The `Debugging` layer would like to call the "execution behavior" implemented by the base method defined in `Programming`. Base methods in layers greatly enhance modularity because they extend classes' interface (e.g., the editor's class is extended in `Programming`) that is accessed by the other layer (e.g., `Debugging`) without modifying the original class definition.

Some formal calculi have been proposed to support such an extension, e.g., (1) requiring a depended layer (i.e. a layer that provides base methods) to be active when the depending layer (a layer that uses those methods) is executing [13] and (2) activating the depended layer on-demand when the

[1]In this paper, we focus on layer-based COP languages. Considerations of our approach under other COP languages such as Subjective-C [10] and Korz [22] are reserved as future work.

COP'15 July 05 2015, Prague, Czech Republic

ACM 978-1-4503-3654-3/15/07 ...$15.00
DOI: http://dx.doi.org/10.1145/2786545.2786550

depending layer is executing [15], and they are proven to be type-sound.[2]

However, these approaches cannot work with the *asynchronous* layer activation [10, 11, 16, 4, 21] in a type-safe manner. The method lookup in existing COP semantics considers all the activated layers and the class of the method receiver. This semantics does not raise a problem when layers are activated using `with`-blocks where corresponding layer activation is synchronous with the currently executing block; however, it raises a problem in asynchronous layer activation where layers are activated and deactivated by external events such as changes in the external environment and operations by the user. These events may occur at any program execution points. Thus, it is possible that the layer providing a method to currently executing method is eventually deactivated, resulting in a method-not-understood error.

In this paper, we propose an another method lookup mechanism for type-safe layer deactivation. In our mechanism, methods are searched in the layer *where the method invocation is placed, and the layers on which that layer depends*, as well as currently activated layers. This inclusion of the "lexical scope" for method lookup addresses the problem of method safety mentioned above. This mechanism is implemented in ServalCJ [19], a COP language that supports asynchronous layer activation as well as synchronous layer activation.

The rest of this paper is structured as follows. Section 2 overviews COP mechanisms such as layers, layer activation, and base methods in a layer. This section also identifies the problem that is tackled in this paper. Section 3 illustrates our proposal of type-safe method lookup for COP. Section 4 compares our proposal with existing approaches. Section 5 discusses the related work. Finally, Section 6 concludes this paper.

2. REVIEWING COP MECHANISMS

We show a motivating example of adaptive user interface, which comprises a text editor program that is inspired by the program editor example shown in [2]. Our example includes a class `Editor` (as well as other classes) to represent an editor view for the user. As shown below, this user interface provides a menubar (as well as other widgets), which is shown by calling `showMenuBar`.

```
class Editor {
  JMenu menu;
  ...
  void showMenuBar() { menu.revalidate(); }
}
```

2.1 Layers and Partial Methods

We consider an additional feature to support programming by using this editor. This feature adaptively becomes available with respect to the type of a file opened by that editor and is dynamically composed with the system. In COP, such a dynamically composed feature is implemented using a layer. The following `Programming` layer implements this feature:

[2]Precisely, there is a flaw in the proof of type soundness for the on-demand activation [15]. To guarantee type soundness, we need to modify the reduction of method invocation to enclose the entire method execution within the activation of all the required layers.

```
layer Programming {
  class Editor {
    JMenuItem start = .., stop = .., resume = ..;
    void showMenuBar() {
      menu.add(start);
      menu.add(stop);
      menu.add(resume);
      proceed();
    }
  }
  /* other (partial) class declarations */
}
```

A *partial method* `showMenuBar` overrides the *base method* declared in `Editor` when `Programming` is *active* (i.e., when it is dynamically composed with the application, which is called layer activation). The `proceed` call invokes the overridden behavior.

In ContextJ [1], the following `with`-block is used for layer activation.

```
Editor editor = new Editor();
with Programming { editor.showMenuBar(); }
```

The layer activation is effective in the *dynamic extent* of the `with`-block. Thus, the `Programming` layer is active when `showMenuBar` is called and thus the partial method defined in `Programming` (that adds several menu items for controlling program execution) is called.

2.2 A Base Method in a Layer

In some COP languages like JCop [3], a layer may also declare a base method, as shown in the following example:

```
layer Programming {
  class Editor {
    void execute() { .. }
    JTextArea getConsole() { .. }
    .. /* same as above */
  }
}
```

In this example, two base methods `execute` and `getConsole` are declared in the layer `Programming`. These methods are not visible from the base program. Thus, the primary purpose of a base method declared in a layer is that it is called from the same layer (or other layers depending on that layer). For example, the `execute` method defines the behavior to start the execution of program. This behavior is registered as an action associated with the menu item added by `Programming` and this is not visible from the base program.

We note that a base method declared in a layer is not just an auxiliary method visible only within that layer; sometimes, it should be visible from other layers. For example, in the adaptive user interface example, we may also consider another additional feature of debugging of the currently developing program. This feature is implemented by the layer `Debugging` that is shown in Figure 1. This layer declares two partial methods, `showMenuBar` and `execute`, and the latter calls `getConsole`. The `getConsole` method is added by `Programming`, which means that `Debugging` assumes the existence of `Programming` and in Figure 1, this dependency is denoted by the `requires` clause in the first line of the layer declaration.

```
layer Debugging requires Programming {
  class Editor {
    JMenuItem stopDebugging = ..;
    void showMenuBar() {
       // a menu item for stopping debugging
      menu.add(stopDebugging);
      proceed();
    }
    void execute() {
      .. /* enabling the step-by-step execution */
      .. getConsole() ..
     /* accessing console to display debug info */
    }
  }
}
```

Figure 1: Layer dependency

This `requires` clause is first introduced by ContextFJ [13] and means that, when `Debugging` is activated, it is necessary that `Programming` is also activated. To activate `Debugging`, we need to activate `Programming` before, which means that `Debugging` can be activated only within the `with`-block that activates `Programming`.

```
Editor editor = new Editor();
with Programming {
  with Debugging { editor.execute(); }
}
```

Within `with Debugging`, both `Programming` and `Debugging` are active, and the partial methods in `Debugging` override the ones in `Programming` because `Debugging` is the most recently activated layer. Thus, the above `execute` call safely calls the `getConsole` provided by `Programming`. The activation of `Debugging` that is not enclosed within the activation of `Programming` results in a compile error in ContextFJ.

Problem. The existing method lookup semantics in COP, where only all the activated layers and the class of the method receiver are considered, cannot describe *asynchronous* layer (de)activation, where layer activation does not enclosed within the statically-known `with`-blocks, in a type-safe manner.[3] Unlike `with`-blocks where layer activation is synchronous with the currently executing block, in asynchronous layer activation, layer activation and deactivation are triggered by external events such as changes in the external environment and operations by the user; i.e., layer (de)activation may occur at any program execution points. For example, in Figure 1, deactivation of `Programming` may occur just before the call of `proceed` in `execute`, resulting a method-not-understood error because this method is introduced by `Programming`.

Supporting base methods in a layer in the asynchronous layer activation is important. Actually, a number of COP languages that provide *asynchronous* layer activation have been proposed [10, 11, 16, 4, 21]. Asynchronous layer activation is useful in a number of COP applications such as ubiquitous computing applications and adaptive user interfaces. The program editor example used in this paper also falls into this category, because events that activate layers are generated by the user's operations.

Besides ContextFJ, there exists a couple of approaches supporting base methods in a layer. However, all these approaches do not address the above issue. We discuss the existing approaches in Section 4.

[3]Moreover, ContextFJ does not support layer deactivation (i.e., `without`) at all.

3. SAFE METHOD LOOKUP FOR COP

There are a couple of approaches to tackle the aforementioned problem. We may prohibit the layer deactivation when it is not safe and postpone the deactivation when it becomes safe. If we take this approach, we need to significantly change the underlying dynamic semantics provided by ContextFJ. Another approach is to change the way of method lookup to avoid the method-not-understood error where only changes in the method lookup is required and the dynamic semantics and the proof of type soundness should be almost identical to those in ContextFJ.

We decided to apply the latter approach because it requires less effort. The idea is to use the enclosing layer for method lookup, i.e., for the method lookup, not only currently activated layers and the base class, but also the layer where the method invocation is written and the layers on which that layer depends are considered.

We formalize the idea as a simple calculus based on ContextFJ [13], whose syntax is provided as follows:

$$
\begin{array}{lcl}
\texttt{CL} & ::= & \texttt{class C} \lhd \texttt{C \{ } \overline{\texttt{C f}}\texttt{; K } \overline{\texttt{M}}\texttt{ \}} \\
\texttt{K} & ::= & \texttt{C(}\overline{\texttt{C f}}\texttt{)\{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}} \\
\texttt{M} & ::= & \texttt{C m(}\overline{\texttt{C x}}\texttt{)\{ return e; \}} \\
\texttt{e,d} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)@X} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)} \\
 & & \mid \texttt{proceed(}\overline{\texttt{e}}\texttt{)} \mid \texttt{v<C,}\overline{\texttt{L}}\texttt{,}\overline{\texttt{L}}\texttt{>.m(}\overline{\texttt{v}}\texttt{)} \\
\texttt{v,w} & ::= & \texttt{new C(}\overline{\texttt{v}}\texttt{)} \\
\texttt{X} & ::= & \texttt{L} \mid \cdot
\end{array}
$$

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 to a method definition. The runtime expression `new C(`$\overline{\texttt{v}}$`)<C,`$\overline{\texttt{L}'}$`,`$\overline{\texttt{L}}$`>.m(`$\overline{\texttt{e}}$`)`, where $\overline{\texttt{L}'}$ is assumed to be a prefix of $\overline{\texttt{L}}$, means that `m` is going to be invoked on `new C(`$\overline{\texttt{v}}$`)`. The annotation `<C,`$\overline{\texttt{L}'}$`,`$\overline{\texttt{L}}$`>` indicates the cursor where method lookup should start.

A method invocation is annotated with a lexical context `X`. It is assumed that if `C m(`$\overline{\texttt{C x}}$`) { return e; }`$\in PT(\texttt{D},\texttt{L},\texttt{m})$ and `e`$_0$`.m`$_1$`(`$\overline{\texttt{e}}$`)@X` is a subexpression of `e`, then `X` $= \texttt{L}$. Similarly, if `C m(`$\overline{\texttt{C x}}$`) { return e; }` is not defined in PT and defined in some class `D`, and `e`$_0$`.m`$_1$`(`$\overline{\texttt{e}}$`)@X` is a subexpression of `e`, then `X` $= \cdot$.

The dependency between layers is modeled by a binary relation $\mathcal{R}$ on layer names; $(\texttt{L}_1, \texttt{L}_2) \in \mathcal{R}$ intuitively means that $\texttt{L}_1$ `requires` $\texttt{L}_2$. We assume a fixed dependency relation and write `L req` Λ, read "layer `L` requires layers Λ," when $\Lambda = \{\texttt{L}' \mid (\texttt{L},\texttt{L}') \in \mathcal{R}\}$. We define two auxiliary functions, *requires* and *filter*, which calculates transitive closure of `req` and removes duplication of layer names, respectively:

$$
\frac{\overline{\texttt{L}}\ \texttt{req}\ \Lambda \qquad \mathit{requires}(\Lambda); \overline{\texttt{L}} = \overline{\texttt{L}'}}{\mathit{requires}(\overline{\texttt{L}}) = \overline{\texttt{L}'}} \qquad \frac{\overline{\texttt{L}}\ \texttt{req}\ \emptyset}{\mathit{requires}(\overline{\texttt{L}}) = \overline{\texttt{L}}}
$$

$$
\begin{array}{rcll}
\mathit{filter}(\overline{\texttt{L}}) & = & \overline{\texttt{L}} & \text{if } \forall \texttt{L}_1, \texttt{L}_2 \in \overline{\texttt{L}}, \\
 & & & \texttt{L}_1 \neq \texttt{L}_2 \\
\mathit{filter}(\overline{\texttt{L}}; \texttt{L}; \overline{\texttt{L}'}; \texttt{L}; \overline{\texttt{L}''}) & = & \mathit{filter}(\overline{\texttt{L}}; \texttt{L}; \overline{\texttt{L}'}; \overline{\texttt{L}''}) & \text{otherwise}
\end{array}
$$

A program $(CT, PT, \texttt{e})$ consists of a class table CT (that

maps a class name to a class definition, as in ContextFJ), a partial method table PT, and an expression `e` that corresponds to the body of the main method. We assume CT and PT are fixed and satisfy some sanity conditions such as: for any `C` in the domain of CT, $CT(\mathtt{C})$ is defined; for any $(\mathtt{m},\mathtt{C},\mathtt{L})$ in the domain of PT, $PT(\mathtt{m},\mathtt{C},\mathtt{L})$ is defined; and there are no cycles in the transitive closure of $\lhd$ (`extends`).

We define the method lookup function $mbody(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}_1,\overline{\mathtt{L}}_2)$ that returns a pair $\overline{\mathtt{x}}.\mathtt{e}$ of parameters and an expression of method `m` in class `C` when the search starts from the sequence of layers $\overline{\mathtt{L}}_1$. $\overline{\mathtt{L}}_2$ keep track of the layers that are active when the search initially started. It also returns the name of class and the sequence of layers where the method has been found, which will be used in reduction rules to deal with `proceed`. *mbody* is defined by following four rules:

$$\frac{\texttt{class C} \lhd \texttt{D \{ .. C}_0\ \texttt{m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\texttt{\{ return e; \} .. \}}}{mbody(\mathtt{m},\mathtt{C},\bullet,\overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \texttt{in C},\bullet}$$

$$\frac{PT(\mathtt{m},\mathtt{C},\mathtt{L}_0) = \mathtt{C}_0\ \texttt{m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\texttt{\{ return e; \}}}{mbody(\mathtt{m},\mathtt{C},(\overline{\mathtt{L}}';\mathtt{L}_0),\overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \texttt{in C},(\overline{\mathtt{L}}';\mathtt{L}_0)}$$

$$\frac{\texttt{class C} \lhd \texttt{D \{ .. M \}} \quad \mathtt{m} \notin \overline{\mathtt{M}} \qquad mbody(\mathtt{m},\mathtt{D},\overline{\mathtt{L}},\overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \texttt{in E},\overline{\mathtt{L}}'}{mbody(\mathtt{m},\mathtt{C},\bullet,\overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \texttt{in E},\overline{\mathtt{L}}'}$$

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

We show the underlying operational semantics where this method lookup is used. This operational semantics is given by a reduction relation of the form $\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e}' \mid \overline{\mathtt{L}}'$, which is read "expression `e` under activated layers $\overline{\mathtt{L}}$ reduces to $\mathtt{e}'$ under $\overline{\mathtt{L}}'$." In particular, formal semantics for method invocation is specified by the following three reduction rules.

$$\frac{\overline{\mathtt{L}}' = \mathit{filter}(\overline{\mathtt{L}};\mathit{requires}(\mathtt{L});\mathtt{L})}{\texttt{new C}(\overline{\texttt{w}})\texttt{.m}(\overline{\texttt{v}})\texttt{@L} \mid \overline{\mathtt{L}} \longrightarrow \texttt{new C}(\overline{\texttt{w}})\texttt{<C,}\overline{\mathtt{L}}'\texttt{,}\overline{\mathtt{L}}'\texttt{>.m}(\overline{\texttt{v}}) \mid \overline{\mathtt{L}}}$$

This first rule is important; it ensures that, besides currently activated layers $\overline{\mathtt{L}}$, the lexical context `L` is always considered when the method is called. This rule is for method invocation where the cursor of the method lookup has not been "initialized"; the cursor is set to be at the receiver's class and the sequence of layers computed by *filter* and *requires* using the currently activated layers $\overline{\mathtt{L}}$ and the layer `L` attached to that method invocation.

The following two rules are straightforward adaptation of the method invocation on the runtime expression of the form `new C(`$\overline{\texttt{v}}$`)<C,`$\overline{\mathtt{L}}$`,`$\overline{\mathtt{L}}$`>.m(`$\overline{\texttt{v}}$`)` from ContextFJ:

$$\frac{mbody(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}'',\overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}\ \texttt{in C}',\bullet \qquad \texttt{class C}' \lhd \texttt{D \{..\}}}{\texttt{new C}_0(\overline{\texttt{v}})\texttt{<C,}\overline{\mathtt{L}}''\texttt{,}\overline{\mathtt{L}}'\texttt{>.m}(\overline{\texttt{w}}) \mid \overline{\mathtt{L}} \longrightarrow \left[\begin{array}{ll}\texttt{new C}_0(\overline{\texttt{v}}) & \texttt{/this},\\ \overline{\texttt{w}} & /\overline{\texttt{x}},\end{array}\right] \mathtt{e} \mid \overline{\mathtt{L}}}$$

In this rule, the receiver is `new C(`$\overline{\texttt{v}}$`)` and the location of the cursor is `<C'`,$\overline{\mathtt{L}}''$`,`$\overline{\mathtt{L}}'$`>`. When the method body is found in the base-layer class $\mathtt{C}'$ (denoted by "`in C'`, $\bullet$"), the whole expression reduces to the method body where the formal parameters $\overline{\mathtt{x}}$ and `this` are replaced with the actual arguments $\overline{\mathtt{w}}$ and the receiver, respectively.

$$\frac{mbody(\mathtt{m},\mathtt{C},\overline{\mathtt{L}}'',\overline{\mathtt{L}}') = \overline{\mathtt{x}}.\mathtt{e}\ \texttt{in C}',(\overline{\mathtt{L}}''';\mathtt{L}_0) \qquad \texttt{class C}' \lhd \texttt{D \{..\}}}{\texttt{new C}_0(\overline{\texttt{v}})\texttt{<C,}\overline{\mathtt{L}}''\texttt{,}\overline{\mathtt{L}}'\texttt{>.m}(\overline{\texttt{w}}) \mid \overline{\mathtt{L}} \longrightarrow \left[\begin{array}{ll}\texttt{new C}_0(\overline{\texttt{v}}) & \texttt{/this},\\ \overline{\texttt{w}} & /\overline{\texttt{x}},\\ \texttt{new C}_0(\overline{\texttt{v}})\texttt{<C'},\overline{\mathtt{L}}'''\texttt{,}\overline{\mathtt{L}}'\texttt{>.m/proceed} & \end{array}\right] \mathtt{e} \mid \overline{\mathtt{L}}}$$

This rule deals with the case where the method body is found in layer $\mathtt{L}_0$ in class $\mathtt{C}'$. In this case, `proceed` in the method body is replaced with the invocation of the same method, where the receiver's cursor points to the next layers $\overline{\mathtt{L}}'''$.

To discuss how this method lookup mechanism works safely with asynchronous layer (de)activation, we also introduce two reduction rules representing layer activation and deactivation, which occur non-deterministically:

$$\frac{f(\mathtt{L},\overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad f = \mathit{activate}\ \text{or}\ \mathit{deactivate}}{\mathtt{e} \mid \overline{\mathtt{L}} \longrightarrow \mathtt{e} \mid \overline{\mathtt{L}}'}$$

where *activate* and *deactivate* are defined as follows:

$$\begin{array}{rcll} \mathit{activate}(\mathtt{L},\overline{\mathtt{L}}) & = & \overline{\mathtt{L}};\mathtt{L} & \text{if } \mathtt{L} \notin \overline{\mathtt{L}} \\ \mathit{activate}(\mathtt{L},\overline{\mathtt{L}}';\mathtt{L};\overline{\mathtt{L}}'') & = & \overline{\mathtt{L}}';\overline{\mathtt{L}}'';\mathtt{L} & \text{otherwise} \end{array}$$

$$\begin{array}{rcll} \mathit{deactivate}(\mathtt{L},\overline{\mathtt{L}}) & = & \overline{\mathtt{L}} & \text{if } \mathtt{L} \notin \overline{\mathtt{L}} \\ \mathit{deactivate}(\mathtt{L},\overline{\mathtt{L}}';\mathtt{L};\overline{\mathtt{L}}'') & = & \overline{\mathtt{L}}';\overline{\mathtt{L}}'' & \text{otherwise} \end{array}$$

The function *activate* put the specified layer `L` at the rightmost position of the activated layer $\overline{\mathtt{L}}$; if `L` is already in $\overline{\mathtt{L}}$, this function changes the order of activated layers so as to the most recently activated layer has the highest priority. The function *deactivate* just removes the specified layer `L` (if exists) from the currently activated layers $\overline{\mathtt{L}}$.

Example. Suppose the situation where `execute` is called on an instance of `Editor` with activated layer `Debugging`, and `Debugging` is asynchronously deactivated during the execution:

```
new Editor().execute()|Debugging
  (method body in Debugging is dispatched)
  ⟶* new Editor().getConsole()@Debugging|Debugging
  (Debugging is deactivated)
  ⟶ new Editor().getConsole()@Debugging
  (method body is found in filter(requires(Debugging)))
  ⟶* {the body of getConsole in Editor in Programming}
  ⟶ ···
```

These reductions demonstrate that the call of `getConsole`, which is written in the layer `Debugging`, succeeds even though `Debugging` is not active when that method is called, illustrating the type safety of our calculus.

Implementation. We implemented this mechanism in ServalCJ [19], a COP language with a generalized layer activation mechanism.[4] Due to the limited space, we do not discuss the implementation details in this paper.

4. DISCUSSION

One traditional question about COP is how to react deactivation of a layer whose partial method is currently executing. Most COP languages adopt the so-called *loyal strategy*

[4] `https://github.com/ServalCJ/pl`

[9], which ensures the completion of the partial method execution and thus the making effect of the deactivation is postponed. Our approach is considered as a natural extension of this strategy: the execution of the required behavior is also ensured when executing the partial method in the requiring layer. Furthermore, our approach ensures the execution of the required layer even when it is not activated at all. In this sense, our approach is also considered as a type-safe application of *on-demand activation* [15] to the asynchronous layer deactivation.

However, our approach raises a subtle problem when the layer deactivation is hard-wired within the layer declaration itself:

```
1 class C { Object n() { ... } }
2 layer L {
3   class C{
4     Object m() { return without L this.n(); }
5     Object n() { ... }
6   }
7 }
```

The class `C` provides the method `n`, which is overridden in the layer `L`. The class `C` in `L` introduces the method `m`, which deactivate `L` and calls `n`. We expect that this call results in the body specified in line 1 because `L` is intentionally deactivated; however, in our mechanism, the definition in line 5 is selected to execute, because the layer where the call of `n` is written, which is `L`, is always considered for the method lookup.

The same issue arises in ServalCJ when we want to ensure deactivation of some layer in some specific control flow [20], which is not possible in ServalCJ. Possibly we may consider an annotation on that layer to declare that this layer's deactivation is ensured but this layer cannot introduce a base method, though this mechanism is reserved as future work.

In the followings, we compare our approach with other existing approaches to provide base methods in layers.

4.1 ContextFJ

We already introduced the ContextFJ approach in Section 2.2. One problem in the ContextFJ approach is that it does not interact nicely with dynamic layer deactivation. In fact, ContextFJ does not support `without`. If layers can dynamically be deactivated, the invocation of a method introduced by the deactivated layer results in a failure when the layer depending on the deactivated layer calls that method. To statically check such an error, we need to gather information about "which layer is absent" at each deactivation point, which is not be very easy, especially in the open-world setting. In short, our mechanism is a simple way to support base methods in a layer in COP languages with dynamic and asynchronous layer deactivation in a type-safe manner.

Nevertheless, we do not argue that `requires` in ContextFJ should be replaced with our mechanism. In particular, the `requires` in ContextFJ would exert its usefulness on requiring the *interface* of layers (although the `requires` in [13] requires the *implementation* of the specified layers.) In ContextFJ, we may assume that the layers providing this interface are active but do not have to concern about the concrete implementations. Likewise, we may write the `requires` clause like "`requires LayerA` or `LayerB`." On the other hand, our mechanism always requires implementations, because the required layers are used for lookup of method bodies.

4.2 On-Demand Activation

On-demand activation [15] has been proposed to avoid the absent layer checking when `requires` is used with `without`. Instead of requiring that the depended layers have been activated, it implicitly activates layers on which currently activated layer depends when these layers are required, as specified by the following `activates` clause:

```
layer Debugging activates Programming {
  /* The body is the same as above */ }
```

We can activate `Debugging` anywhere, regardless of the condition whether this activation is enclosed with the activation of `Programming`; if `Programming` is not active, it is implicitly activated when currently executing behavior requires that.

This mechanism implicitly activates the layer specified by `requires` when that layer is asynchronously deactivated. However, this mechanism fails if currently executing layer is deactivated, which is explained in the following code:

```
layer L {
  class C {
    Object m() {  // deactivating L
      return this.n(); }
    Object n() {  // introduced by L
      return new Object(); }
  }
}
```

The class `C` in layer `L` defines partial methods `m` and `n`, and `m` calls `n`, which is introduced by `L`. Since asynchronous layer deactivation may occur at any execution points in the on-demand activation mechanism, it is possible that the deactivation of `L` occurs just before the call of `n`. Then, the method lookup for `n` is performed without the existence of `L`, leading to the method-not-understood error.

4.3 Layer Inheritance

Our approach is considerably similar with the layer inheritance approach; i.e., the `requires` relations look like the `extends` relations between layers. JCop [3] supports such layer inheritance mechanism.

Currently there are no formal semantics for layer inheritance and the type-safety of layer inheritance w.r.t. the asynchronous layer deactivation is not clear.[5] In JCop, layers are instantiated, and layers are not directly activated but instances of layers are. If an instance `p1` of `Programming` and instance `d1` of `Debugging` are activated, the partial method defined in `Programming` is executed once for `p1` and once for `d1`, which would result in multiply displaying the same menu items for the programming feature. On the other hand, our approach is based on the COP model where a layer is not a first-class citizen, and prohibits such a duplication.

5. RELATED WORK

Inoue et al. discuss a safe type system for JCop [14]. Basically, it is an application of type system developed in ContextFJ; however, it also supports layer inheritance and first-class layers. It does not deal with layer deactivation directly; instead, it supports an idiom, which is called layer swapping, for layer deactivation. It is prohibited to deactivate layers

[5] JCop basically provides synchronous layer (de)activation.

using `without`; however, we may "swap" a layer with other one, which is compatible with the swapped layer.

Dependency between layers may also be represented in the form of *composite layers* [7, 18]. In [7], an extension of ContextL [8] with layer composition operators such as and-composition and or-composition. At each layer activation point, it calculates the set of depended layers and activates them. If that set is ambiguous, it suspends the execution until when the user resolves this ambiguity. In [18], a similar mechanism is discussed in [18] under event-based layer transition [16]. FECJ° [17] formalizes the operational semantics of composite layers implemented in EventCJ [16]. The dependency between layers can also be specified in some COP languages such as Subjective-C [10] and Ambience [11]. In these languages, such dependency is checked at runtime.

Our mechanism is similar with Newspeak's method lookup [6] in that the lexical scope is used for method lookup. While Newspeak uses the lexical scope to resolve method collisions between the outer class and the super classes, our mechanism uses the lexical scope (and the definitions on which that scope depends) as a safety-net ensuring that at least there is a behavior while executing the current method, even when the enclosing layer is immediately deactivated. Furthermore, our method lookup includes the dynamically activated layers, and we need to define the appropriate ordering between those activated layers, the layer comprising the lexical scope, the layers on which that scope depends, and the base class.

6. CONCLUDING REMARKS

This paper addresses the problems of allowing layers to declare base methods when these base methods are used with asynchronous layer deactivation. This paper provides a formal definition of method lookup that uses the lexical scope of a method invocation, and informally demonstrates how this mechanism solves the problem. This mechanism is considered as a type-safe application of on-demand activation mechanism to the asynchronous layer deactivation that addresses the problem of ContextFJ in supporting dynamic layer deactivation. This mechanism is also considered as an alternative to the layer inheritance mechanism where duplicated calls of partial methods do not occur at all. This mechanism is implemented in our COP language, ServalCJ.

7. REFERENCES

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

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

[3] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *SC'10*, volume 6144 of *LNCS*, pages 50–65, 2010.

[4] 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.

[5] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The role object pattern. In *PLoP'97*, 1997.

[6] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *ECOOP'10*, pages 405–428, 2010.

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

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

[9] Brecht Desmet, Jorge Vallejos, Pascal Costanza, and Robert Hirschfeld. Layered design approach for context-aware systems. In *VaMoS'07*, 2007.

[10] 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.

[11] 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.

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

[13] Atsushi Igarashi, Robert Hirschfeld, and Hidehiko Masuhara. A type system for dynamic layer composition. In *FOOL'12*, 2012.

[14] Hiroaki Inoue, Atsushi Igarashi, Malte Appeltauer, and Robert Hirschfeld. Towards type-safe JCop: A type system for layer inheritance and first-class layers. In *COP'14*, 2014.

[15] Tetsuo Kamina, Tomoyuki Aotani, and Atsushi Igarashi. On-demand layer activation for type-safe deactivation. In *COP'14*, 2014.

[16] 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.

[17] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A core calculus of composite layers. In *FOAL'13*, pages 7–12, 2013.

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

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

[20] Jens Lincke, Robert Krahn, and Robert Hirschfeld. Implementing scoped method tracing with ContextJS. In *COP'11*, 2011.

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

[22] David Ungar, Harold Ossher, and Doug Kimelman. Korz: Simple, symmetric, subjective, context-oriented programming. In *Onward! 2014*, pages 113–131, 2014.