A Core Calculus of Composite Layers

Tetsuo Kamina
University of Tokyo
kamina@acm.org

Tomoyuki Aotani
Japan Advanced Institute of Science and Technology
aotani@jaist.ac.jp

Hidehiko Masuhara
University of Tokyo
masuhara@acm.org

Abstract

Composite layers in context-oriented programming (COP) are the abstraction that localizes conditions about when the specified layer becomes active. A composite layer changes the behavior of the system by *implicit layer activation* triggered by explicit activation of contexts. Existing studies on formalization of COP languages do not cover such an activation mechanism. This paper formalizes composite layers to clarify the operational semantics of implicit layer activation. Based on this formalization, we prove that the translation of composite layers into the existing COP language is sound, which ensures the correctness of the existing implementation of composite layers.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages

Keywords Context-oriented programming; EventCJ; Implicit layer activation

1. Introduction

Composite layers [10, 11] in context-oriented programming (COP) [7] are the abstraction that consolidates context-dependent behavior executable only under some condition specified in terms of contexts. This mechanism localizes such a condition about when the layer becomes active, which simplifies the program that would be complicated without using this mechanism in particular when the correspondence between contexts and variations of behavior is not simple.

Composite layers change the behavior of the system by *implicit layer activation*, which is triggered by the explicit activation of contexts. On the other hand, existing formalizations for COP languages only provide explicit layer activation in their operational semantics [8, 1]. This limitation makes it hard to investigate some important properties on composite layers. For example, in the previous paper [11], we proposed translation of composite layers into the existing COP language, namely EventCJ [9], because the implicit layer activation mechanism would require additional computation overhead. The lack of formal studies on composite layers makes it difficult to prove the soundness of this translation.

FOAL'13, March 26, 2013, Fukuoka, Japan.
Copyright © 2013 ACM 978-1-4503-1865-5/13/03. . . $15.00

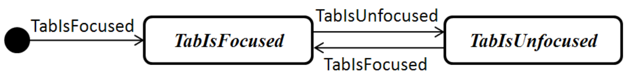

Figure 1. Context changes in the Twitter client. The black circles indicate "initial contexts" in which the system resides when it is initialized

To clarify the operational semantics of composite layers, in this paper, we propose a formalization of composite layers, called FECJ$^\circ$, as an extension of Featherweight EventCJ (FECJ), a small calculus that models EventCJ's layer transition mechanism [1]. To model the implicit layer activation, we extend FECJ to include the *activate declaration table (AT)*, which maps each composite layer to a condition about when the layer is active. We also modify the reduction rules to encode the activation of composite layers into the calculus. Although this extension is simple, it clarifies two significant facts with respect to the application order of layer activation. First, explicit activation of layers and implicit activation of composite layers are not applied simultaneously. Second, unlike FECJ, in FECJ$^\circ$, the order of the resulting active layers cannot be determined by the semantics.

Based on this formalization, we obtained the following technical results:

- We prove that the composite layer mechanism does not go wrong; i.e., layers that do not satisfy the condition for activation never become active.
- We prove that the compilation of composite layers described in [11] is sound. The compilation translates activation of composite layers into layer transition rules in EventCJ. We formalize this compilation as a translation from FECJ$^\circ$ into FECJ, and prove that both are behaviorally equivalent.

This paper is organized as follows. Section 2 provides a limitation of layer-based COP languages and introduces composite layers. Section 3 formally describes operational semantics of composite layers. Section 4 describes a formalization of the implementation. Section 5 discusses related work. Finally, Section 6 concludes this paper.

2. Preliminaries

An Example In this section, we explain our motivation by considering a tabbed Twitter client. This Twitter client is equipped with multiple tabs, each of which displays the user's timeline; the timeline is updated after a person followed by the user posts a tweet. Only one tab is focused at a time. The focused tab frequently updates the timeline, while the other unfocused tabs infrequently update it. The user selects a tab by clicking on it.

Context changes are modeled by using a state transition model, as shown in Figure 1. Besides an initial state, this model declares

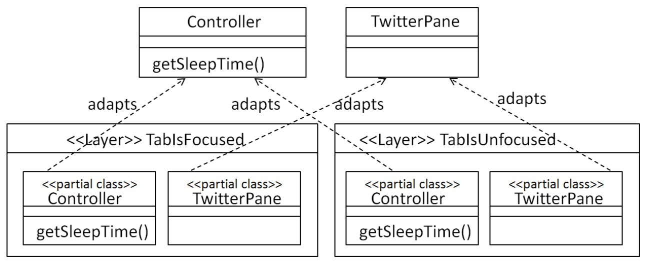

Figure 2. Classes and layers in COP languages.

```
event TabIsFocused(ChangeEvent e)
  :after execution(void TabListener.stateChanged(*))
  :sendTo(e.getSrc().getSelected().controller());

transition TabIsFocused:
  TabIsFocused ? TabIsUnfocused -> TabIsFocused
| -> TabIsFocused;
```

Figure 3. Examples of an event and a layer transition rule in EventCJ

two states, namely *TabIsFocused* and *TabIsUnfocused*. The former represents a context when the tab is focused, while the latter represents a context when the tab is unfocused. Behavior of the application changes with respect to current contexts. When the tab is focused, the timeline is frequently updated, and when the tab is unfocused, it is infrequently updated.

Layer-based COP languages provide layer to modularize context-dependent behaviors and a mechanism to dynamically switch layers. A layer is a modularization unit that groups behaviors executable under the same contexts. Figure 2 illustrates classes and layers in COP languages by using a class diagram, as proposed by Lincke et al.[13]. In this diagram, a layer is represented as a container stereotyped with `<<Layer>>`. Each layer declares *partial methods* that change the original behavior when the layer is *active*. These partial methods are grouped into a class stereotyped with `<<partial class>>`. For example, when `getSleepTime` is called, the partial method `getSleepTime` declared in the layer `TabIsFocused` is executed when `TabIsFocused` is active.

In Layer-based COP languages, we *explicitly* specify layers that are activated/deactivated at particular execution points. For example, EventCJ [9] provides layer transition rules that specify the layers to be activated (and deactivated) upon a specific event, as shown in Figure 3. An event is declared with two specifications, one indicating when the event is generated and the other indicating where the event is sent. The former is specified by using an AspectJ-like pointcut sublanguage [12], and the latter is specified using the `sendTo` clause that lists instances where the event is sent. For example, `TabIsFocused` is generated when the focus of a tab is changed and sent to the focused tab. Upon events, EventCJ executes layer transition rules for changing the active layers of the object that receives the events. The layer transition rule shown in Figure 3 is read as, "upon the generation of `TabIsFocused`, if `TabIsUnfocused` is active, then it is deactivated and `TabIsFocused` is activated; otherwise, only `TabIsFocused` is activated."

Problem on Layer-based COP Languages Assume that the Twitter client is extended to include the *energy-saved* mode, which is a context when the executing machine is running out of battery. In this context, any tab (including the focused one) updates the time-

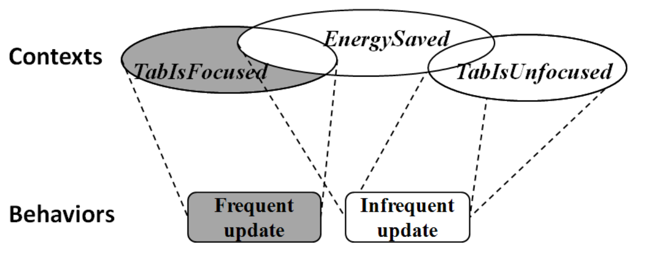

Figure 4. Correspondence between contexts and behaviors

line infrequently to save resources, and an alert icon is displayed to notify the user about the current status of the battery.

This extension implies that the behaviors implemented in layers `TabIsFocused` and `TabIsUnfocused` no longer depend solely on the contexts *TabIsFocused* and *TabIsUnfocused*, respectively. Instead, they depend on combinations of contexts relating to a tab's focus and the machine's battery. The correspondence between contexts and behaviors is shown in Figure 4.

Since the condition corresponding to the layer being active is changed, we need to modify the event shown in Figure 3 as follows:

```
event TabIsFocused(ChangeEvent e)
  :after execution(void TabListener.stateChanged(*))
    &&args(e)&&if(!Env.isBatteryLow())
  :sendTo(e.getSrc().getSelected().controller());
```

In this extension, there is a tangling code problem; we need to declare events that depend on several context changes and status, because each context change in Figure 1 no longer directly corresponds to a layer switching. For example, when the tab's context changes to *TabIsFocused*, the layer `TabIsActive` becomes active only when the system is not in the context *EnergySaved*. Thus, in EventCJ, we have to declare the event that activates `TabIsActive` with the joinpoint that changes the tab's focus and the `if` pointcut that inspects the status of the battery. This event does not directly reflect the model of context changes. Two context changes relating to the tab's focus and the status of the battery are tangled in the same event. Thus, the resulting code makes it difficult to perform further extension and modification.

Furthermore, there are some cases where we need to declare several events for the same context change, since each context change does not correspond to a unique event. For example, for the context change from the *EnergySaved* to the initial state (i.e., the system is not in the energy-saved mode), we need to declare two events: an event generated when the tab is in *TabIsUnfocused* and that generated when the tab is in the initial state. This duplication of events requires the layer transition rules for each event; this requirement makes the program complicated and results in it being poorly understood.

Composite Layers To tackle the aforementioned problem, we proposed *composite layers* [10, 11]. A composite layer depends on activation of other layers. It declares a proposition in which ground terms are other layer names (true when active), and implicitly becomes active only when the proposition holds. We call layers that are not composite layers as atomic layers. Atomic layers are explicitly activated by using layer transition rules.

For example, we can declare two composite layers corresponding to two variations of behavior in Figure 4 as follows:

```
layer FrequentUpdate
    when TabIsFocused && !EnergySaved {
  /* frequent update of timeline */
}
```

```
layer InfrequentUpdate
    when TabIsUnfocused || EnergySaved {
  /* infrequent update of timeline */
}
```

Both composite layers FrequentUpdate and InfrequentUpdate depend on activation of TabIsFocused, TabIsUnfocused, and EnergySaved, which are controlled by the following layer activation rules:

```
transition TabIsFocused:
  TabIsUnfocused ? TabIsUnfocused -> TabIsFocused
| -> TabIsFocused;

transition TabIsUnfocused:
  TabIsFocused ? TabIsFocused -> TabIsUnfocused
| -> TabIsUnfocused;

transition BatteryLevelLow: -> EnergySaved;

transition ACConnected: EnergySaved ->;
```

Composite layers solve the tangling problem, because there is a one-to-one correspondence between atomic layers and contexts so that context changes are straightforwardly implemented by layer transition rules. While designing and implementing events and context transition rules about the status of the tab, we do not have to consider the status of battery. Thus, the proposal enhances separation of concerns.

3. Formalization

The introduction of the implicit layer activation mechanism requires the modification of operational semantics of EventCJ. The operational semantics of EventCJ was given as a small calculus called Featherweight EventCJ (FECJ) [1], which is built on top of ContextFJ [8]. This section shows FECJ$^\circ$, an extension of FECJ, to reflect the implicit layer activation mechanism.

In short, we extend the syntax of FECJ to include the *activate declaration table* (*AT*) that maps each composite layer to a condition about when the layer is active. We also modify the reduction rules to incorporate the *AT*. Although this modification is simple, it reveals two significant facts. First, layer transition rules and implicit activation are not applied simultaneously. Instead, implicit activation is applied just after all the applicable transition rules are applied. Second, unlike FECJ, in FECJ$^\circ$ the order of the resulting active layers after the application of transition rules and implicit activation cannot be determined by the semantics. We clarify these points by formally defining the operational semantics.

Note that this formalization only focuses on the operational semantics. A formal study on the type system of FECJ$^\circ$ remains as future work.

Syntax The abstract syntax of FECJ$^\circ$ is shown in Figure 5. Let metavariables C, D, and E range over class names; L ranges over layer names; f ranges over field names; m ranges over method names; ℓ ranges over labels; c ranges over conditions; v and w range over addresses in stores; and x and y range over variables, which include a special variable this. Overlines denote sequences: e.g., $\overline{\mathtt{f}}$ stands for a possibly empty sequence $\mathtt{f}_1, \cdots, \mathtt{f}_n$. The empty set is denoted by $\emptyset$. The empty sequence is denoted by $\bullet$ and sometimes by $\emptyset$ if the order is not important. We use a set in a context where a sequence is expected (e.g., comparing a set and a sequence using $=$). In this case, we regard the set as a sequence obtained by serializing all elements in the set in an arbitrary order. We also abbreviate a sequence of pairs by writing "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$" for "$\mathtt{C}_1\ \mathtt{f}_1, \cdots, \mathtt{C}_n\ \mathtt{f}_n$," where n denotes the length of $\overline{\mathtt{C}}$

$$
\begin{array}{lcll}
\mathtt{CL} & ::= & \mathtt{class\ C} \lhd \mathtt{C\ \{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\ \mathtt{\}} & (\textit{classes}) \\
\mathtt{K} & ::= & & (\textit{constructors}) \\
 & & \mathtt{C(}\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{)\{\ super(}\overline{\mathtt{f}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\ \}} & \\
\mathtt{M} & ::= & \mathtt{C\ m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} & (\textit{methods}) \\
\mathtt{e}, \mathtt{d} & ::= & \mathtt{x} \mid \mathtt{e}^{\hat{\ell}}\mathtt{.f} \mid \mathtt{e}^{\hat{\ell}}\mathtt{.m(}\overline{\mathtt{e}^{\hat{\ell}}}\mathtt{)} & (\textit{expressions}) \\
 & & \mid \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} & \\
 & & \mid \mathtt{proceed(}\overline{\mathtt{e}}\mathtt{)} & \\
 & & \mid \mathtt{v} \mid \mathtt{v<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{v}}\mathtt{)} & \\
\mathtt{t} & ::= & \overline{\mathtt{L}}\mathtt{:}\overline{\mathtt{L}}\mathtt{?}\overline{\mathtt{L}}\mathtt{\rightarrow}\overline{\mathtt{L}} & (\textit{transitions}) \\
\mathtt{c} & ::= & \mathtt{L} \mid \mathtt{!L} \mid \mathtt{c||c} \mid \mathtt{c\&\&c} & (\textit{conditions}) \\
\mathtt{p} & ::= & \mathtt{v} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\overline{\mathtt{L}}\mathtt{>} & (\textit{partial stores}) \\
\mu & ::= & \overline{\mathtt{p}} & (\textit{stores})
\end{array}
$$

Figure 5. FECJ$^\circ$: abstract syntax

and $\overline{\mathtt{f}}$. Similarly, we write "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}$" as shorthand for the sequence of declarations "$\mathtt{C}_1\ \mathtt{f}_1\mathtt{;}\ldots\mathtt{C}_n\ \mathtt{f}_n\mathtt{;}$" and "$\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;}$" as shorthand for "$\mathtt{this.f}_1\mathtt{=f}_1\mathtt{;}\ldots\mathtt{;this.f}_n\mathtt{=f}_n\mathtt{;}$". We use commas and semicolons for concatenations. 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 a label ℓ or no labels. An empty element is denoted by ϵ, which is usually omitted.

The only addition to the syntax from [1] that we make in this paper is conditions, which correspond to conditions in composite layers. A condition can be a layer name L, its negation !L, logical-or c||c, and logical-and c&&c. Other details are exactly the same as [1].

An FECJ$^\circ$ program $(\mathit{CT}, \mathit{PT}, \mathit{TT}, \mathit{AT}, \mathtt{e})$ consists of a class table *CT* that maps a class name to a class definition, a partial method table *PT* that maps a triple C, L, and m of class, layer, and method names to a method definition, a transition rule table *TT* that maps a label to a sequence of transition rules, an activation table *AT* that is a set of pairs of a layer and a condition, and a well-formed expression e that corresponds to the body of the main method. We assume *CT*, *PT*, *TT*, and *AT* to be fixed and to satisfy the following sanity conditions:

1. $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C\ \ldots}$ for any $\mathtt{C} \in \mathit{dom}(\mathit{CT})$.
2. $\mathtt{Object} \notin \mathit{dom}(\mathit{CT})$.
3. For every class name C (except Object) appearing anywhere in *CT*, we have $\mathtt{C} \in \mathit{dom}(\mathit{CT})$;
4. There are no cycles in the transitive closure of the extends clauses.
5. $\mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L}) = \mathtt{\ldots\ m(\ldots)\{\ldots\}}$ for any $(\mathtt{m}, \mathtt{C}, \mathtt{L}) \in \mathit{dom}(\mathit{PT})$.
6. $\mathit{TT}(\ell) = \overline{\mathtt{t}}$ for every label ℓ that appears in e, *CT*, and *PT*.
7. For every $(\mathtt{L}, \mathtt{c}) \in \mathit{AT}$, $\mathit{TT}(\ell) = \overline{\mathtt{L}}_1\mathtt{:}\overline{\mathtt{L}}_2\mathtt{?}\overline{\mathtt{L}}_3 \rightarrow \overline{\mathtt{L}}_4$ does not contain L, and $\forall \mathtt{L}' \in \mathtt{c}, \mathtt{L}' \notin \mathit{dom}(\mathit{AT})$.

Operational Semantics The operational semantics of FECJ$^\circ$ is given by a reduction relation of the form $\mathtt{e}|\mu \longrightarrow \mathtt{e}'|\mu'$, which is read "expression e under the store μ reduces to $\mathtt{e}'$ under μ'." We assume that μ and μ' do not contain duplicate names.

We only show the rule that is different from FECJ, which is a rule for the application of transition rules.

$$
\frac{\mathit{switchLayer}(\mu, \mathtt{v}, \ell) = \mathtt{p}}{J[\mathtt{v}^{\ell}]|\mu \longrightarrow J[\mathtt{v}]|\mathtt{p}\mu} \qquad \text{(R-LABEL)}
$$

$$\frac{\mu(\mathtt{v}) = \mathtt{new\ C(\overline{w})<...>} \qquad \mathit{transit}(\mu, \mathtt{v}, \ell) = \overline{\mathtt{L}} \qquad \mathit{layer}(\overline{\mathtt{L}}) = \overline{\mathtt{L}}' \qquad \overline{\mathtt{L}} \oplus \overline{\mathtt{L}}' = \overline{\mathtt{L}}''}{\mathit{switchLayer}(\mu, \mathtt{v}, \ell) = \mathtt{v} \mapsto \mathtt{new\ C(\overline{w})<}\overline{\mathtt{L}}''\mathtt{>}} \quad (\textsc{Tr-SwitchLayer})$$

$$\frac{\begin{array}{c}\mu(\mathtt{v}) = \mathtt{new\ C(\overline{w})<}\overline{\mathtt{L}}\mathtt{>} \\ \mathit{TT}(\ell) = \overline{\mathtt{t}} \qquad \{\mathtt{t} \in \overline{\mathtt{t}} | \mathit{app}(\mathtt{t}, \overline{\mathtt{L}})\} = \overline{\mathtt{t}}' \\ \oplus\{\overline{\mathtt{L}}_3 | \overline{\mathtt{L}}_1 : \overline{\mathtt{L}}_2 ? \overline{\mathtt{L}}_3 \rightarrow \overline{\mathtt{L}}_4 \in \overline{\mathtt{t}}'\} = \overline{\mathtt{L}}' \\ \oplus\{\overline{\mathtt{L}}_4 | \overline{\mathtt{L}}_1 : \overline{\mathtt{L}}_2 ? \overline{\mathtt{L}}_3 \rightarrow \overline{\mathtt{L}}_4 \in \overline{\mathtt{t}}'\} = \overline{\mathtt{L}}'' \\ \overline{\mathtt{L}}''' = (\overline{\mathtt{L}} \setminus \overline{\mathtt{L}}' \setminus \mathit{dom}(\mathit{AT})) \oplus \overline{\mathtt{L}}''\end{array}}{\mathit{transit}(\mu, \mathtt{v}, \ell) = \overline{\mathtt{L}}'''} \quad (\textsc{Tr-Transition})$$

$$\frac{\overline{\mathtt{L}}_1 \cap \overline{\mathtt{L}} = \overline{\mathtt{L}}_1 \qquad \overline{\mathtt{L}}_2 \cap \overline{\mathtt{L}} = \bullet}{\mathit{app}(\overline{\mathtt{L}}_1 : \overline{\mathtt{L}}_2 ? \overline{\mathtt{L}}_3 \rightarrow \overline{\mathtt{L}}_4, \overline{\mathtt{L}})} \quad (\textsc{Pred-Applicable})$$

$$\frac{\overline{\mathtt{L}'} = \{\mathtt{L} \in \mathit{dom}(\mathit{AT}) | \overline{\mathtt{L}} \vdash \mathit{AT}(\mathtt{L})\}}{\mathit{layer}(\overline{\mathtt{L}}) = \overline{\mathtt{L}}'} \quad (\textsc{Tr-Active})$$

Figure 6. Transition rules

The form $J[\cdot]$ represents evaluation contexts for operations defined as follows:

$$J ::= [\,]\mathtt{.m(\overline{v})} \mid \mathtt{v.m(\overline{w},}[\,]\mathtt{,\overline{w}')} \mid [\,]\mathtt{.f}$$

Each context is an expression with a hole (written $[\,]$) somewhere inside it. We write $J[\mathtt{v}]$ for either field access or method invocation obtained by replacing the hole in J with $\mathtt{v}$, which reflects the fact that each expression is well-formed. The function *switchLayer*, which is explained later, applies the transition rules for the label $\hat{\ell}$ and updates the active layers for $\mathtt{v}$ by evaluating conditions stored in *AT*.

Transitions The function *switchLayer* changes the set of active layers on an object by applying transition rules (Figure 6). The actual application of transition rules is performed by the function *transit* that returns a sequence of active layers after the application of transition rules. These active layers are subjected to the application of the activate declarations in *AT*, by the function *layer*. Note that *layer* uses the result of *transit*, i.e., *layer* is applicable only after *transit*.

The definition of *transit* is also shown in Figure 6. We use set-like notations to denote the operations on sequences; all the operations preserve the order of the sequences. $\{\mathtt{t} \in \mathtt{t}_1, .., \mathtt{t}_n | \mathit{pred}(t)\}$ is the sequence $\mathtt{t}_1', .., \mathtt{t}_k'$, where $\mathtt{t}_1'$ and $\mathtt{t}_k'$ are the first and the last elements in $\mathtt{t}_1, ..., \mathtt{t}_n$ that satisfy the predicate *pred*; further, for all $\mathtt{t}_i$ and $\mathtt{t}_j$ in $\mathtt{t}_1', .., \mathtt{t}_k'$ and $i < j$, there exist $\mathtt{t}_{i'}$ and $\mathtt{t}_{j'}$ in $\mathtt{t}_1, .., \mathtt{t}_n$ that satisfy $\mathtt{t}_i = \mathtt{t}_{i'}$, $\mathtt{t}_j = \mathtt{t}_{j'}$, and $i' < j'$. $\overline{\mathtt{L}} \setminus \overline{\mathtt{L}}'$ indicates the operation that removes all the elements in the sequence $\overline{\mathtt{L}}'$ from $\overline{\mathtt{L}}$ without changing the order of $\overline{\mathtt{L}}$. $\overline{\mathtt{L}} \oplus \overline{\mathtt{L}}'$ concatenates the two sequences without any duplications, i.e., $(\mathtt{L}_1; ..; \mathtt{L}_i; ..; \mathtt{L}_n) \oplus (\mathtt{L}_1'; ..; \mathtt{L}_i'; \mathtt{L}_i; \mathtt{L}_{i+2}'; ..; \mathtt{L}_k')$ is the sequence $\mathtt{L}_1; ..; \mathtt{L}_i; ..; \mathtt{L}_n; \mathtt{L}_1'; ..; \mathtt{L}_i'; \mathtt{L}_{i+2}'; ..; \mathtt{L}_k'$. We write $\oplus\{\overline{\mathtt{L}}, .., \overline{\mathtt{L}}'\}$ as shorthand for $\overline{\mathtt{L}} \oplus\ ..\ \oplus \overline{\mathtt{L}}'$. $\overline{\mathtt{L}} \cap \overline{\mathtt{L}}'$ denotes the set-intersection between $\overline{\mathtt{L}}'$ and $\overline{\mathtt{L}}$ that preserves the order of layers in $\overline{\mathtt{L}}$ and $\overline{\mathtt{L}}$.

The function *transit* takes a store μ, location $\mathtt{v}$, and label ℓ. Unlike the original definition in [1], it returns a sequence of active layers, which are the result of the application of transition rules for ℓ. The transitions are obtained by looking up the transition table *TT*. The predicate $\mathit{app}(\overline{\mathtt{L}}_1 : \overline{\mathtt{L}}_2 ? \overline{\mathtt{L}}_3 \rightarrow \overline{\mathtt{L}}_4, \overline{\mathtt{L}})$ means that the transition $\mathtt{t}$ satisfies its precondition, i.e., all the layers in $\overline{\mathtt{L}}_1$ are in $\overline{\mathtt{L}}$ and none of the layers in $\overline{\mathtt{L}}_2$ is in $\overline{\mathtt{L}}$. Layers to be deactivated and activated are determined from the transitions. In other words, all the transitions on the same label are applied at the same time.

The function *layer* takes a sequence of layers $\overline{\mathtt{L}}$ and evaluates all the conditions in *AT* by assuming that all $\overline{\mathtt{L}}$ are true. Then, it returns a sequence of layers in the range of *AT* where the condition is evaluated to be true. $\overline{\mathtt{L}} \vdash \mathit{AT}(\mathtt{L})$ is read "the condition $\mathit{AT}(\mathtt{L})$ is true when every $\mathtt{L} \in \overline{\mathtt{L}}$ is active." The evaluation of a condition $\mathtt{c}$ is performed by interpreting `!`, `||`, and `&&` as negation, logical-or, and logical-and of propositions, respectively. Unlike the definition of *transit*, *layer* does not determine the order of the resulting layers, i.e., the resulting sequence is obtained by serializing all elements in $\{\mathtt{L} \in \mathit{dom}(\mathit{AT}) | \overline{\mathtt{L}} \vdash \mathit{AT}(\mathtt{L})\}$ in an arbitrary order.

Properties From the definitions described in this section, we can prove the following simple theorem ensuring that "layers that do not satisfy the condition for activation never become active."

THEOREM 3.1. *If* $\mathtt{e}|\emptyset \longrightarrow^* \mathtt{e}'|\mu$*,* $\mathtt{v} \mapsto \mathtt{new\ C(\overline{v})<}\overline{\mathtt{L}}\mathtt{>} \in \mu$ *and* $\mathtt{L} \in \mathit{dom}(\mathit{AT})$*, then* $\overline{\mathtt{L}} \vdash \mathit{AT}(\mathtt{L})$ *iff* $\mathtt{L} \in \overline{\mathtt{L}}$*.*

PROOF. See Appendix A.

4. Implementation

Our implementation strategy is to translate a program written in the extended syntax into that written in the original syntax. Specifically, in the extended syntax, layer switching is represented in terms of context changes. We need to translate them into layer transition rules that directly activate and deactivate layers. In fact, this translation algorithm is identical to that of composite layers [11]. In our previous work, however, there are no proof showing that the translation preserves behavior. In this section, we formulate the translation algorithm more concisely by using the set notations, and prove that FECJ$^\circ$ and FECJ [1] (obtained by the translation algorithm) are behaviorally equivalent. To prove this behavioral equivalence, it is enough to prove that active layers obtained by evaluating conditions stored in *AT* after applying the applicable transition rules are exactly the same as those obtained by applying the applicable transition rules under the translated layer transition rules.

Before showing the translation, we need to explain some preprocessing that are required to perform the translation. The translation shown in [11] firstly constructs a state transition diagram (as shown in Figure 1) from the layer transition rules; i.e., we translate *TT* into the canonical form where $\mathtt{t.1} = \mathtt{t.3}$ and $\mathtt{t.2} = \emptyset$ for all $\mathtt{t} \in \mathit{TT}$ (we write the nth element of $\mathtt{t}$ in *TT* as $\mathtt{t.n}$). The translation then constructs a parallel composition of *TT*.

Now, we formulate this translation as a compilation from FECJ$^\circ$ into FECJ, which translates *AT* and *TT* into layer transition rules in FECJ, say TT', as follows:

```
Let $\mathcal{L}$ be the set of all contexts within the program.
$\mathit{TT}'(\ell) =$
    let $\mathit{TT}^c(\ell)$   $=$   $\{\overline{\mathtt{L}} : \bullet\ ?\overline{\mathtt{L}} \rightarrow \overline{\mathtt{L}} \setminus \overline{\mathtt{t}}.3 \oplus \overline{\mathtt{t}}.4 | \overline{\mathtt{L}} \in \Pi 2^{\mathcal{L}},$
                        $\overline{\mathtt{t}} = \{\mathtt{t} \in \mathit{TT}(\ell) | \mathit{app}(\mathtt{t}, \overline{\mathtt{L}})\} \neq \emptyset\}$
     in
    let $\mathit{equiv}(x)$   $=$   $\bigoplus\{y \in 2^{\mathcal{L}} | \mathit{layer}(x) \subseteq \mathit{layer}(y)\}$ in
    $\mathit{TT}(\ell) \oplus \{\mathit{layer}(\mathtt{t}.1) : (\mathit{TT}^c(\ell).1 \setminus \mathtt{t}.1) \cap \mathit{equiv}(\mathtt{t}.1)?$
                $\mathit{layer}(\mathtt{t}.3) \rightarrow \mathit{layer}(\mathtt{t}.4) | \mathtt{t} \in \mathit{TT}^c(\ell)\}$
```

Intuitively, $\mathit{TT}^c(\ell)$ is a superset of parallel composition of *TT*; it consists of all combinations of transition rules that can be constructed over $\mathcal{L}$, restricting the fourth element to be filtered by rules in *TT* that are applicable on given $\overline{\mathtt{L}}$. This formulation is performed to make the proof easy. Since every rule $\mathtt{t} \in \mathit{TT}^c(\ell)$ that are not in the parallel composition of *TT* never become applicable during computation, this formulation does not affect the result of compilation.

The result of compilation is a union of $\mathit{TT}(\ell)$ (transition rules in the FECJ$^\circ$ program) and the layer transition rules constructed from

AT by assigning layers that are active on the condition when $\mathtt{t}.n$ is active. These layers are obtained by evaluating the *layer* function. The second element in the resulting transition rule is a guard to preserve the original behavior of implicit layer activation (we write $\mathit{TT}^c(\ell)\mathtt{.1}$ as a shorthand of $\bigoplus\{\mathtt{t.1}|\mathtt{t} \in \mathit{TT}^c(\ell)\}$). This guard is required, because activation of layers occurs only under specific condition of contexts.

Now, we formulate the theorem ensuring that the set of layers obtained by TR-SWITCHLAYER and that obtained by applying layer transition rules under TT' are identical .

THEOREM 4.1. *If* $J[\mathtt{v}^\ell]|\mu \longrightarrow_{\mathrm{FECJ}^\circ} J[\mathtt{v}]|\mathtt{p}\mu$ *and* $J[\mathtt{v}^\ell]|\mu \longrightarrow_{\mathrm{FECJ}} J[\mathtt{v}]|\mathtt{p}'\mu$ *where* $\longrightarrow_{\mathrm{FECJ}}$ *is evaluated under* TT'*, then* $\mathtt{p} = \mathtt{p}'$ *for some* $\mathtt{v}$*,* ℓ*, and* μ*.*

PROOF. See Appendix B.

Example: the composite layers and transition rules shown in Section 2 is translated into the following layer transition rules:

```
// transition rules from the original program
transition Focusing:
  TabIsUnfocused ? TabIsUnfocused -> TabIsFocused
| -> TabIsFocused;
transition Unfocusing:
  TabIsFocused ? TabIsFocused -> TabIsUnfocused
| -> TabIsUnfocused;
transition BatteryLow: -> EnergySaved;
transition ACConnected: EnergySaved ->;
// generated transition rules
transition Focusing:
  InfrequentUpdate,!EnergySaved ?
    InfrequentUpdate -> FrequentUpdate
  -> FrequentUpdate;
transition Unfocusing
  FrequentUpdate ?
    FrequentUpdate -> InfrequentUpdate
  -> InfrequentUpdate;
transition BatteryLow:
  FrequentUpdate ?
    FrequentUpdate -> InfrequentUpdate
  -> InfrequentUpdate;
transition ACConnected:
  InfrequentUpdate,!TabIsUnfocused ?
    InfrequentUpdate -> FrequentUpdate
  -> FrequentUpdate;
transition ACConnected:
  InfrequentUpdate,!TabIsFocused,!TabIsUnfocused ?
     InfrequentUpdate ->;
```

5. Related Work

Costanza and D'Hondt have proposed a method analyzing the dependency between layers; the method involves the use of feature diagrams [2], where each feature is mapped onto a layer. They provide an extension of ContextL [3] to represent composite layers (layers that correspond to composite features). Since their method can represent relations between layers, for example, "layer `A` includes one of layers `B`, `C`, and `D`" and/or "layer `X` includes all layers `Y` and `Z`," it shares similarities with our approach. The difference is that their approach provides explicit activation of composite layers, while other layers (depending on the activated layers) are automatically activated. Although explicit activation of composite layers is sometimes convenient when we know that all the constituent layers are active on some parts of the base program, it does not solve the problems addressed in this paper, because the problems require not the automatic composition of layers but flexible mapping from contexts to layers.

Tanter et al. proposed context-aware aspects [14], which are aspects whose behaviors depend on contexts. This concept is realized as a framework where a context is defined as a pointcut, which is similar to AspectJ's `if` pointcut while also being capable of restricting the *past* contexts. Contexts are composable, because they are realized as pointcuts.

There are several COP languages that are not layer-based. Interestingly, these languages share some similarities with the extended version of EventCJ proposed in this paper. For example, Ambience [6] and its successor AmOS [5] are prototype-based context-oriented languages featuring multimethods and subjective dispatch. Unlike layer-based COP languages, a context is reified as an object that is implicitly argumented to the method invocation. Thus, each context-dependent method is defined with a context, which can be a combined context that is similar to the composition of contexts in the `when` clause of activate declarations. While an activate declaration can be declared with an arbitrary proposition, Ambience and AmOS support only intersections. Subjective-C [4] is an extension of Objective-C with context-orientation concepts. Like Ambience and AmOS, in Subjective-C, context-dependent behaviors are defined for each method using the `#context` annotation that specifies a context on which the method depends. It provides a small domain-specific language (DSL) to represent relations between contexts. Our proposal, on the other hand, puts more emphasis on grouping behaviors that are executable under the same context. Thus, our approach is still *layer-based* and it has the advantages of aforementioned languages by bridging between contexts and layers.

6. Concluding Remarks

In this paper, we presented an formalization of composite layers. This clarifies some significant facts about the operational semantics of composite layers such as the evaluation order between layer transition rules and propositions from composite layers, and the order of active layers. The proofs provided by this paper ensure that the composite layer mechanism does not go wrong and the translation-based implementation is sound.

References

[1] 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.

[2] Pascal Costanza and Theo D'Hondt. Feature descriptions for context-oriented programming. In *2nd International Workshop on Dynamic Software Product Lines (DSPL'08)*, 2008.

[3] 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.

[4] 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.

[5] 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.

[6] Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *DLS'07*, pages 77–88, 2007.

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