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.

To clarify the operational semantics of composite layers, in this paper, we propose a formalization of composite layers, called FECJ\textsuperscript{2}, 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\textsuperscript{2}, 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\textsuperscript{2} 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
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 timeline 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:

```java
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 TabIsFocused 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:

```java
layer FrequentUpdate
    when TabIsFocused && !EnergySaved {
        /* frequent 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 ? TabIsUnfocused → TabIsUnfocused
| → TabIsUnfocused;

transition BatteryLevelLow: → EnergySaved;

transition AConnected: EnergySaved →;

Compositional 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 re-
quires 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\*, an extension of FECJ, to
reflect the implicit layer activation mechanism.

In short, we extend the syntax of FECJ to include the activate
declarations table (AT) that maps each composite layer to a condi-
tion 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\* 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\* remains
as future work.

Syntax The abstract syntax of FECJ\* 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; \( \ell \) ranges over labels; \( e \) 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.,
\( \overline{\tau} \) stands for a possibly empty sequence \( f_1, \ldots, f_n \). The empty
set is denoted by \( \emptyset \). The empty sequence is denoted by \( \epsilon \) 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 se-
quence obtained by serializing all elements in the set in an ar-
bitrary order. We also abbreviate a sequence of pairs by writing
\( \overline{C, \overline{\tau}} \) for \( \{C_1, f_1, \ldots, C_n, f_n\} \), where \( n \) denotes the length of \( \overline{C} \)
and \( \overline{\tau} \). Similarly, we write \( \overline{\tau, \overline{\tau}} \) as shorthand for the sequence
of declarations \( C_1, f_1; \ldots; C_n, f_n\) and \( \text{this.}\overline{\tau} = \overline{\tau} \) as shorthand
for \( \text{this.f}_1 = f_1; \ldots; \text{this.f}_n = f_n \).

We use commas and semi-
colons for concatenations. It is assumed that sequences of field dec-

\[
\begin{align*}
\text{CL} &::= \text{class } C \triangleq C \left\{ \overline{\tau} ; \overline{\tau} \right\} \\
\text{K} &::= \text{class } C \left\{ \overline{\tau} ; \overline{\tau} \right\} \\
\text{M} &::= \text{m} \left\{ \overline{\tau} \right\} \\
\text{e.d} &::= x \mid e \cdot f_e \cdot e \cdot m \left( e \right) \\
\text{t} &::= \text{t} \mid \text{t} \mid \text{t} \rightarrow \text{t} \\
\text{c} &::= L \mid L \mid L \mid L \mid L \mid L \mid L \mid L \mid L \mid L \\
\text{p} &::= v \rightarrow \text{new} \left\{ \overline{\tau} \right\} L \left( \overline{\tau} \right) \\
\mu &::= \overline{\mu} \\
\end{align*}
\]

Figure 5. FECJ\*: abstract syntax

The operational semantics of FECJ\* is given by a reduction relation of
the form \( e\mu \rightarrow e'\mu' \), which is read “expression \( e \) under the store \( \mu \) reduces to \( e' \) under \( \mu' \).” We assume
that \( \mu \) and \( \mu' \) 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.

\[
\begin{align*}
\text{switchLayer}(\mu, v, \ell) &\rightarrow p & \text{(R-LABEL)}
\end{align*}
\]
The function layer takes a sequence of layers \( L \) and evaluates all the conditions in \( AT \) by assuming that all \( L \) are true. Then, it returns a sequence of layers in the range of \( AT \) where the condition is evaluated to be true. \( \bar{L} \vdash AT(L) \) is read "the condition \( AT(L) \) is true when every \( L \) in \( \bar{L} \) is active." The evaluation of a condition \( c \) is performed by interpreting \( t, \ell, \) and \( k \) 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 \( \{ L \in dom(AT) \mid \bar{L} \vdash AT(L) \} \) 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 \( e \mid \emptyset \rightarrow^* e' \mid \mu, \nu \rightarrow new C(\tau) \in D \in \mu \) and \( L \in dom(AT) \), then \( L \vdash AT(L) \) iff \( L \in \bar{L} \).

**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 interpret 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 FEC\( \in \) 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 \( t_1 = t_2 = \emptyset \) for all \( t \in TT \) (we write the nth element of \( t \) in \( TT \) as \( .n \)). The translation then constructs a parallel composition of \( TT \).

Now, we formulate this translation as a compilation from FEC\( \in \) into FECJ, which translates \( AT \) and \( TT \) into layer transition rules in FECJ, say \( TT' \), as follows:

Let \( L \) be the set of all contexts within the program.

\[
TT'(\ell) = \begin{cases} \{ L : \tau L \rightarrow L \in C, \tau \notin C \} & \ell \in \Pi 2^C, \\ \{ \ell : \tau \in TT(\ell) \mid app(t, L) \neq \emptyset \} & \end{cases}
\]

Intuitively, \( TT'(\ell) \) is a superset of parallel composition of \( TT \); it consists of all combinations of transition rules that can be constructed over \( L \), restricting the fourth element to be filtered by rules in \( TT \) that are applicable on given \( L \). This formulation is performed to make the proof easy. Since every rule \( t \in TT'(\ell) \) 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 \( TT'(\ell) \) (translation rules in the FECJ program) and the layer transition rules constructed from
AT by assigning layers that are active on the condition when \( t \cdot n \) 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 \( TT' (\ell) \)). As a shorthand of \( \bigoplus \{ t.1 | t \in TT' (\ell) \} \). 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 \( J[v] \mid \mu \boldsymbol{\rightarrow}_{FECL} J[v] \mid p \mu \) and \( J[v'] \mid \mu \boldsymbol{\rightarrow}_{FECL} J[v] \mid p' \mu \) where \( \bigoplus_{FECL} \) is evaluated under \( TT' \), then \( p = p' \) for some \( \nu, \ell, \) and \( \mu \).

**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 ? TabIsFocused -> TabIsFocused;
 transition Unfocusing: TabIsFocused ? TabIsFocused -> TabIsUnfocused
 | -> TabIsUnfocused;
 transition BatteryLow: -> TabIsUnfocused;
 transition ACConnected: EnergySaved ->;
```

**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 refined 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**


Before giving the proof, we develop a number of required lemmas. We write \( \mathcal{E}, n \) as a shorthand for \( \bigoplus \{ t, n \mid t \in \mathcal{E} \} \). We write \( |L| \) as a set of all elements in \( L \), and \( \Pi X \) as a set of all permutations for \( X \).

**Lemma B.1.** Let \( \mathcal{L} \) be a sequence of layers satisfying 
\[
layer(\mathcal{L}) = \mathcal{L} \cap dom(AT)
\]
Then, for any \( x \in \Pi 2^\mathcal{L} \), \( layer(x) \subseteq layer(\mathcal{L}) \) holds if 
\[
layer(x) \cap \mathcal{L} = layer(x)
\]

**Proof.**
\[
layer(x) \cap \mathcal{L} = layer(x)
\]
\[
\equiv layer(x) \subseteq \mathcal{L}
\]
(by def. of \( \subseteq \))
\[
\equiv layer(x) \cap dom(AT) \subseteq \mathcal{L} \cap dom(AT)
\]
(monotonicity of \( \cdot \cap dom(AT) \))
\[
\equiv layer(x) \subseteq \mathcal{L} \cap dom(AT)
\]
(since \( layer(x) \subseteq dom(AT) \) by def. of layer)
\[
\equiv layer(x) \subseteq layer(\mathcal{L})
\]
(since \( layer(\mathcal{L}) = \mathcal{L} \cap dom(AT) \))

**Lemma B.2.** Let \( \mathcal{L} \) be a sequence of layers satisfying 
\[
layer(\mathcal{L}) = \mathcal{L} \cap dom(AT)
\]
Then \( \mathcal{L} \setminus dom(AT) \subseteq x \) for any \( x \in 2^\mathcal{L} \) and \( \ell \) if all of the following equations hold:
- \( \exists t \in TT(\ell), t.1 \cap \mathcal{L} = t.1 \wedge t.2 \cap \mathcal{L} = \cdot \)
- \( \exists t \in TT(\ell), t.1 \cap x = t.1 \wedge t.2 \cap x = \cdot \)
- \( layer(x) \cap \mathcal{L} = layer(x) \)
- \( (TT^\ell(1 \setminus x) \cap \equiv(x) \cap \mathcal{L} = \cdot \)

**Proof.**
\[
(\mathcal{L} \setminus dom(AT)) \subseteq x
\]
\[
\equiv (\mathcal{L} \setminus dom(AT)) \setminus dom(TT^\ell(1 \setminus x) \cap \equiv(x) \cap \mathcal{L} = \cdot
\]
\[
\equiv (\mathcal{L} \setminus dom(AT)) \setminus x
\]

**Lemma B.3.** Let \( \mathcal{L} = \mathcal{L} \cap dom(AT) \). Then for any label \( \ell \in \mathcal{L} \cap dom(TT) \), the following equation holds:
\[
\mathcal{L} \setminus \mathcal{L} \setminus dom(AT) \oplus \mathcal{L} \setminus dom(AT) \setminus dom(TT^\ell(1 \setminus x) \cap \equiv(x) \cap \mathcal{L} = \cdot
\]
where \( \mathcal{L}, \mathcal{L}', \mathcal{R} \) and \( \mathcal{R}' \) are:
\[
\mathcal{L} = \bigoplus \{ t.3 \mid t \in TT(\ell), app(t, \mathcal{R}) \}
\]
\[
\mathcal{L}' = \bigoplus \{ t.4 \mid t \in TT(\ell), app(t, \mathcal{R}) \}
\]
\[
\mathcal{R} = \bigoplus \{ t.3 \mid t \in TT(\ell), app(t, \mathcal{L}) \}
\]
\[
\mathcal{R}' = \bigoplus \{ t.4 \mid t \in TT(\ell), app(t, \mathcal{L}) \}
\]

**Proof.** By Lemma B.1 and B.2. □

We are now ready to give the proof of Theorem 4.1, which is immediately given by Lemma B.3. □