IPSJ Transactions on Programming Vol.6 No.1 1–8 (Jan. 2013)

Regular Paper

Introducing Composite Layers in EventCJ

Tetsuo Kamina[1,a)] Tomoyuki Aotani[2,b)] Hidehiko Masuhara[1,c)]

Received: May 1, 2012, Accepted: October 1, 2012

Abstract: Context-oriented programming (COP) languages provide a modularization mechanism called a layer, which modularizes behaviors that are executable under specific contexts, and specify a way to dynamically switch behaviors. However, the correspondence between real-world contexts and units of behavioral variations is not simple. Thus, in existing COP languages, context-related concerns can easily be tangled within a piece of layer activation code. In this paper, we address this problem by introducing a new construct called a composite layer, which declares a proposition in which ground terms are given other layer names (true when active). A composite layer is active only when the proposition is true. We introduce this construct into EventCJ, out COP language, and verify this approach by conducting two case studies involving a context-aware Twitter client and a program editor. The results obtained in our approach show that the layer activation code is simple and free from tangled context-related concerns. We also discuss the efficient implementation of this mechanism in EventCJ.

Keywords: context-oriented programming, atomic layers, activation, languages

1. Introduction

Context-oriented programming (COP) [11] is a research topic that is becoming more intensively studied, as it modularizes variations of behavior that depend on context. A context is an environment in which (a part of) a program is executed, including the external environment of the executing machine, and the states of other parts of the program. For example, in the case of a pedestrian navigation system running on a mobile terminal, the situation can be regarded as a context since the the user is either outdoors or indoors; cases where the remaining battery charge is not low (normal) or low (energy-saving mode) are also considered as contextual information. Several COP languages provide linguistic constructs that modularize variations of behavior that depend on contexts using *layers* and to activate/deactivate them according to the executing contexts [2], [4], [6], [12]. In this paper, we refer to such languages as *layer-based COP languages*.

There are several advantages to layer-based COP languages. First, they enable the separation of crosscutting concerns, as they modularize variations of context-dependent behavior that crosscut existing modules, such as classes, using a layer. Second, they enable the disciplined control of layer activation/deactivation. For example, ContextJ [2] and JCop [4] limit the activation of layers under the control flow starting from the specified block (or method call), which makes it easy to avoid accidental conflicts between variations of behavior. EventCJ [12] provides a mechanism for controlling layer activation based on a state transition model. Thus, it also makes it easy to avoid the accidental conflicts between layers, and it enables the control of layer activation, which is not limited under the specific control flows.

Until now, in layer-based COP languages, it is assumed that a unit of behavior modularized using a layer depends on one context. We can observe this assumption from the fact that, in their layer activation control mechanisms, we explicitly specify activating/deactivating layers when contexts change.

However, since the unit of behavior modularized using a layer does not always correspond to a single context, there is the following problem in the existing layer-based COP languages. From the viewpoint of maintainability, when we model how the contexts change and when this change occurs in the real world, it is desirable that the program separately describes each variation of behavior depending on a context. However, there exist some units of behavior that are executable only under a combination of specific contexts. Thus, in the existing layer-based COP languages, the specifications about which layer is activated and when this activation occurs become complex, and several concerns regarding contexts exist in the layer activation code in a tangled manner.

This paper proposes a new linguistic mechanism *composite layer* and demonstrates how the aforementioned problem is solved by using this mechanism. A composite layer is a layer that depends on the activation of other layers, and declares a proposition in which the ground terms are names of other layers (true when active). A composite layer is active only when this proposition is true.

This paper also proposes how to introduce composite layers into EventCJ, and validates their effectiveness using two case studies: a Twitter client and a program editor [3]. Both case studies show that the use of composite layers simplifies the layer ac-

[1] The University of Tokyo, Bunkyo, Tokyo 113–0033, Japan
[2] Japan Advanced Institute of Science and Technology, Nomi, Ishikawa 923–1292, Japan
[a)] kamina@acm.org
[b)] aotani@jaist.ac.jp
[c)] masuhara@acm.org

Preliminary ideas for the language design were published at the COP'12 workshop [13]. This paper extends the given paper by describing an implementation strategy for the proposed method.

tivation code when compared with the case where they are not used, and concerns regarding contexts are not tangled within the layer activation code.

Furthermore, this paper proposes an implementation strategy for EventCJ with composite layers. In this strategy, the extension of EventCJ with composite layers is translated into (the original version of) EventCJ, which does not have composite layers. Thus, we can implement the proposed extension as preprocessing that does not change the existing EventCJ compiler.

This paper is organized as follows. Section 2 describes the example used throughout this paper. It also describes the existing layer-based COP languages and their problems. Section 3 explains composite layers. Section 4 demonstrates how the aforementioned problems are treated by introducing composite layers into EventCJ. Section 5 discusses how to implement the proposed method. Section 6 describes related work. Finally, Section 7 concludes this paper.

2. Problem Statement

2.1 Example: A Twitter Client

In this section, we describe the problems addressed in this paper using the example of a Twitter client. This system is equipped with multiple tabs, and each tab displays a timeline (a temporal sequence of tweets submitted by the followed users). The system only displays the timeline on the currently-selected tab, and the timelines on other unselected tabs are hidden beneath the selected tab. We cannot select multiple tabs at a time. The timeline on the selected tab is frequently updated; for the effective use of resources, other timelines are infrequently updated. Furthermore, when the machine's power supply is on the verge of running out, the timelines on all tabs are infrequently updated.

There are two kinds of context changes in this system. The first one pertains to the states of the tab, and the other to the remaining battery charge. We may elicit such contexts by using requirements engineering methods (such as [17]), which is outside of the scope of this paper. In **Fig. 1**, we show context changes in the Twitter client described in a state transition diagram. Below, we use round-cornered rectangles to represent states. We use a state with an empty label to represent an initial state.

The behavior of the Twitter client changes with respect to contexts. The variations of behavior that switch with respect to contexts are "frequent update of timeline" and "infrequent update of timeline," as stated above[*1]. In **Fig. 2**, we show the correspondence between contexts and behavioral variations. We observe that these two variations do not depend only on the individual state of each state transition diagram, but also on their combinations.

2.2 Layer-based COP Languages

Most layer-based COP languages modularize variations of behavior by using layers.

[*1] In practice, there are other behavioral variations such as the display of an alert icon to indicate that the remaining battery charge is low, and stopping the display of a user's icon (an image) when the system is running in the energy-saving mode. In general, such context-dependent variations of behavior are crosscutting concerns. However, in this paper, we do not mention this fact for simplicity.

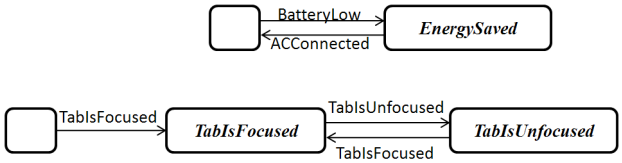

Fig. 1 Context changes in the Twitter client.

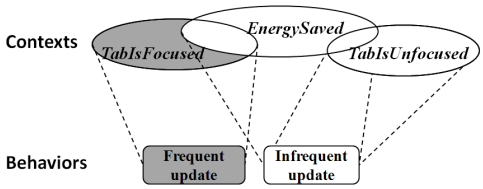

Fig. 2 Correspondence between contexts and variations of behavior.

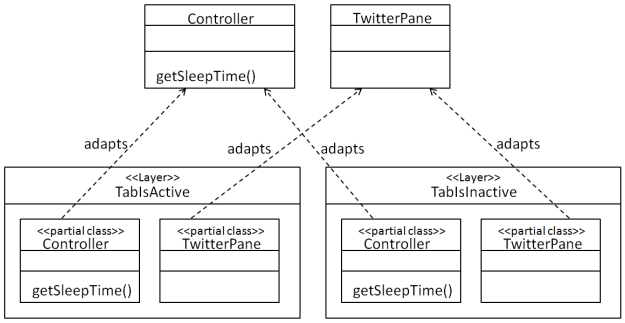

Fig. 3 Relationship between layers and classes.

Figure 3 represents how the variations "frequent update of timeline" and "infrequent update of timeline" are separated using a class diagram; this separation uses layers `TabIsActive` and `TabIsInactive` for each variation, respectively. In this paper, we represent a layer by a container stereotyped with `<<Layer>>`, which is a notation proposed by Ref. [16]. Within a layer, we declare partial methods (in Fig. 3, we represent a set of partial methods as a class stereotyped with `<<partial class>>`). A partial method is executable only when the enclosing layer is active, and it changes the behavior of the class to which the layer is applied (i.e., the class at the target of the dashed arrow in Fig. 3). For example, when `TabIsActive` is active, at the call of `Controller.getSleepTime()`, the partial method `getSleepTime` declared in `TabIsActive` is called, instead of the original method[*2].

Several methods have been proposed for the control of layer activation, such as the specification of a dynamic scope enclosed with a `with`-block [2], [6], and layer activation that is based on events and layer transition rules [12]. Below, we describe the layer activation in EventCJ. In EventCJ, we specify when layers are switched, and on which objects this switching occurs by using the following event declaration:

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

[*2] We may declare a partial method with the modifier `before` or `after`. In this case, the partial method is called just before or after the call of the original method, respectively.

This event declaration specifies when the event TabIsFocused is generated by using the pointcut designators provided by AspectJ [15], which is just after the execution of the void TabListener.stateChanged method. The sendTo clause followed by the pointcut specifies the objects to which the event is sent.

When the object receives the event, it changes the active layers as specified by the layer transition rules, where we directly specify layers to be activated/deactivated using the names of layers. We show an example of a layer transition rule upon the generation of TabIsFocused as follows:

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

This rule concatenates two subrules using the | operator. Each subrule is written in the form of "*Guard*?*Layers1*-> *Layers2*." In *Guard*, we list the names of layers, each of which is interpreted as true when this layer is active. In *Layer1*, we list the layers to be deactivated. In *Layer2*, we list the layers to be activated. When there are no layers to be specified, we leave the corresponding parts empty. The guards in subrules concatenated by | are evaluated from left to right, and only the left-most applicable rule is selected. Thus, the above rule is read as "if TabIsUnfocused is active, then it is deactivated and TabIsFocused is activated; otherwise, just TabIsFocused is activated."

Note that we also directly specify the names of layers to be activated in other layer-based COP languages. For example, in ContextJ [2], we specify the layers to be activated within the dynamic scope of the specified block by using the following with-statement:

```
with(TabIsFocused) { .. }
```

2.3 Problem with the Existing Layer-based COP Languages

As mentioned above, the activation of layers depends on the state changes of multiple contexts in the real world. On the other hand, in the existing layer-based COP languages, we have to explicitly specify layers to be activated when contexts change. Thus, a layer depends on multiple contexts. Thus, in these languages, the specifications about which layers are activated and when this activation occurs become complex. Furthermore, several concerns regarding contexts exist in the layer activation code in a tangled manner. We explain this problem using the case in which we implement the Twitter client using EventCJ.

In EventCJ, we identify the execution point when contexts change by using events. The name of the event corresponds to the label of the edge in Fig. 1; we identify events TabIsFocused and TabIsUnfocused, which change the selection of the tab, and BatteryLow and ACConnected, which change the status of the battery charge. These events change the activation of layers.

Note that changes in layer activation do not correspond to those in the contexts. For example, when TabIsFocused is generated, the context always changes to *TabIsFocused*; however, the layer TabIsActive becomes active only when the system is not in *EnergySaved*. Thus, in EventCJ, we need to declare TabIsFocused as an event that depends on the status of battery charge. For example, assuming that we have a method isBatteryLow that inspects the status of battery charge, we can declare an event that is generated when the result of isBatteryLow is true, by using the if pointcut:

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

There are two disadvantages of this method. First, the model of the real world is not directly reflected in the program. Thus, the information identified in the model is modified. For example, the meaning of TabIsFocused in Fig. 1 is altered in the above event declaration, which makes it difficult to maintain the program when the model is modified. Second, this approach makes the source code complex. For example, in the case where ACConnected is generated, the required transition rule differs according to the state of the tab: *TabIsFocused* or the initial state. Thus, we need to declare different events for each case, and to declare transition rules for each event, which requires additional lines of code.

3. A Proposal of Composite Layers

To tackle the aforementioned problem, we propose the use of composite layers. A composite layer is a layer whose activation depends on the activation of other layers, and declares a proposition where ground terms are names of other layers (true when active). A composite layer is active only when the proposition is true. In this paper, a layer that is not a composite layer (i.e., a layer in the existing layer-based COP languages) is referred to an *atomic layer*.

3.1 Atomic Layers

An atomic layer has a one-to-one correspondence for each state of context changes. In other words, an atomic layer represents a context. If there is a variation of behavior that depends only on the state of context changes (i.e., that does not depend on other states of context changes), we can modularize such a variation using an atomic layer. Since there are no such variations in the example of the Twitter client, we declare every atomic layer with an empty body. In most layer-based COP languages, we declare layers using the keyword layer as follows:

```
layer TabIsFocused {}
layer TabIsUnfocused {}
layer EnergySaved {}
```

We can distinguish the atomic layers from composite layers, because the former do not have a when clause (explained in the next section). Only atomic layers can be directly controlled using layer transition rules and with-blocks.

3.2 Composite Layers

A composite layer is used to modularize a variation of behavior that depends on a combination of multiple contexts. In the example of the Twitter client, we declare the behavior of frequent

updates of the timeline and infrequent updates of the timeline by using composite layers. Each composite layer declares a condition when the layer is active by using a proposition as follows:

```
layer TabIsActive
  when TabIsFocused && !EnergySaved {
    /* Frequent updates of timeline */
}
layer TabIsInactive
  when TabIsUnfocused || EnergySaved {
    /* Infrequent updates of timeline */
}
```

The newly introduced syntax is the `when` clause, which specifies the condition when the layer is active. Within the `when` clause, we use the names of layers, each of which is interpreted as true when the corresponding layer is active. Thus, `TabIsActive` is active only when `TabIsFocused` is true and `EnergySaved` is false. Similarly, `TabIsInactive` is active only when `TabIsUnfocused` or `EnergySaved` is true. We cannot explicitly activate composite layers (by using `with`-blocks or layer transition rules).

4. Introducing Composite Layers into EventCJ and Its Evaluation

We show how the problem described in Section 2 is tackled by introducing composite layers into EventCJ.

First, by specifying only atomic layers in layer transition rules, the model of the real world becomes directly reflected in the program. **Figure 4** is an example of the layer transition rules translated from the state transition diagram shown in Fig. 1. Since each atomic layer corresponds to each state of the state transition diagram, each transition rule corresponds only to one state transition. Thus, there are no such problems, which specified that multiple state transitions are tangled into one layer transition rule. Furthermore, we do not have to use the `if` pointcut to inspect the state of transitions, because we can declare events that have one-to-one correspondence to each label of the state transitions. Thus, we do not have to provide different event declarations and layer transition rules for each state, *TabIsFocused* or the initial state, of the tab. Each variation of the behavior regarding the frequency of updates of the timeline is described in the aforementioned composite layers `TabIsActive` and `TabIsInactive`, and is implicitly activated according to the condition specified by the `when` clause.

In **Table 1**, we show (1) the number of event declarations, (2) the number of subrules of layer transition rules, and (3) the number of layer transition rules where multiple context changes are tangled, for each case where we use composite layers to implement the Twitter client (for the number of tangled rules, we also count the case when an event corresponding to a layer transition rule depends on multiple context changes), and where we do not use them. All comparisons show that the number is smaller when we use composite layers, which indicates that the program has been simplified. Furthermore, there are no layer transition rules where multiple context changes are tangled, which implies that the descriptions separated in the model of the real world are also separated in the program.

```
1  transition TabIsFocused:
2    TabIsUnfocused ? TabIsUnfocused -> TabIsFocused
3  | -> TabIsFocused;
4
5  transition TabIsUnfocused:
6    TabIsFocused ? TabIsFocused -> TabIsUnfocused
7  | -> TabIsUnfocused;
8
9  transition BatteryLevelLow: -> EnergySaved;
10
11 transition ACConnected: EnergySaved ->;
```

Fig. 4 Layer transition rules after introducing composite layers.

Table 1 Comparison between the existing approach and the proposed approach for the Twitter client.

	w/o comp. layers	w/ comp. layers
# of event decl.	5	4
# of subrules	9	6
# of tangling rules	5	0

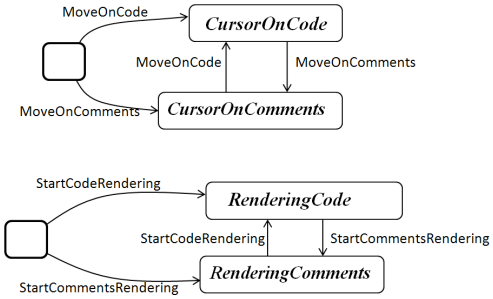

Fig. 5 Context changes on CJEdit.

4.1 Example 2: A Program Editor

We validate the effectiveness of composite layers by using another example. CJEdit [3] is a program editor that enhances the readability of programs by applying different formatting methods for code editing parts and comment editing parts. Code editing parts are displayed in the type-writer format with syntax highlighting. Comment editing parts are displayed in rich text format (RTF), and we can use several fonts, text sizes, decorations, and alignments. Furthermore, the arrangement of GUI components such as the menu bar and text blocks changes according to the position of the cursor (in code editing parts or in comment editing parts).

In CJEdit, we can identify two kinds of context changes with respect to the cursor position and rendering text regions; there is a state for each situation where the cursor is on code (*CursorOnCode*) or on comments (*CursorOnComments*), and each situation where the code is rendered (*RenderingCode*) or comments are rendered (*RenderingComments*), respectively. **Figure 5** shows context changes in CJEdit described in a state transition diagram.

The behavior of CJEdit changes according to changes of contexts. We show context-dependent variations of behavior in CJEdit as follows. Since these variations are modularized into layers, the names of the respective layers in each parenthesis are also displayed.

- To display GUI components for code editing functions (such as an outline view of program structure) when the cursor is on code (`CursorOnCode`)
- To display GUI components for comment editing functions

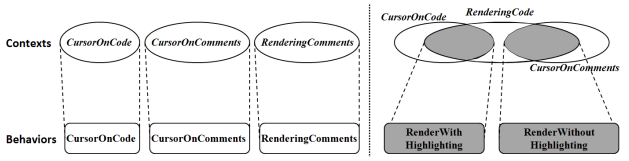

Fig. 6 Correspondence between layers and variations of behaviors on CJEdit.

(such as the menu and tools for specifying fonts, text sizes, decorations, and alignments) when the cursor is on comments (`CursorOnComments`)

- To display the code with syntax highlighting when the cursor is on code (`RenderWithHighlighting`)
- To display the code without syntax highlighting when the cursor is on comments (`RenderWithoutHighlighting`)
- To display comments with RTF (`RenderComments`)

The variations that are selected for execution depend on the states of two state transitions, as in the case of the Twitter client. We show the correspondence between contexts and variations of behavior (layer) in **Fig. 6**. Although each of the three layers `CursorOnCode`, `CursorOnComments`, and `RenderingComments` corresponds to each of the contexts *CursorOnCode*, *CursorOnComments*, and *RenderingComments*, respectively, the remaining two layers depend on two states in the state transition diagram.

The implementation of the CJEdit using EventCJ with composite layers is as follows[*3]. First, we declare three layers that have one-to-one correspondence with the contexts as atomic layers containing definitions of behavior:

```
layer CursorOnCode {
  /* partial methods implementing the
     code editing features */ }
layer CursorOnComments {
  /* partial methods implementing the
     comment editing features */ }
layer RenderingComments {
  /* partial methods for the RTF display */ }
```

On the other hand, we declare the context *RenderingCode* that does not have any one-to-one correspondence to behavior as an atomic layer with an empty body:

```
layer RenderingCode {}
```

We implement the remaining layers as composite layers by using the abovementioned atomic layers:

```
layer RenderWithHighlighting
  when RenderingCode && CursorOnCode {
    /* partial methods for displaying source
       code with syntax highlighting */
}
layer RenderWithoutHighlighting
  when RenderingCode && CursorOnComments {
    /* partial methods for displaying source
       code without syntax highlighting */
}
```

[*3] The case of using EventCJ without composite layers is described in Ref. [14].

```
1  transition MoveOnCode:
2    CursorOnComments ? CursorOnComments -> CursorOnCode
3  | -> CursorOnCode;
4
5  transition MoveOnComments:
6    CursorOnCode ? CursorOnCode -> CursorOnComments
7  | -> CursorOnComments;
8
9  transition StartCodeRendering
10   RenderingComments ? RenderingComments -> RenderingCode
11 | -> RenderingCode;
12
13 transition StartCommentRendering
14   RenderingCode ? RenderingCode -> RenderingComments
15 | -> RenderingComments;
```

Fig. 7 Layer transition rules for CJEdit after introducing composite layers.

Table 2 A comparison between the existing approach and the proposed approach for the program editor.

	w/o comp. layers	w/ comp. layers
# of event decl.	4	4
# of subrules	11	8
# of tangling rules	4	0

Figure 7 is an example of layer transition rules translated from the state transition diagram shown in Fig. 5. As in the case of the Twitter client, there are no such problems, which specified that multiple state transitions are tangled into one layer transition rule, because we specify the activation of only atomic layers in layer transition rules.

Table 2 shows the number of event declarations, that of subrules of layer transition rules, and that of layer transition rules where multiple context changes are tangled, for each scenario whether we use composite layers to implement CJEdit. Although there are no changes in the number of event declarations, the number of subrules is smaller when we use composite layers, which indicates that the program has been simplified. As in the case of Twitter client, there are no layer transition rules where multiple context changes are tangled, which implies that the descriptions separated in the model of the real world are also separated in the program.

5. Discussion on Implementation

We can translate EventCJ with composite layers into the original version of EventCJ without composite layers. The key principle for this translation is to convert composite layers and the control of their activation into layers without `when` clauses and layer transition rules, respectively, and to execute the converted layer transition rules by using existing events and layer transition rules specified by using atomic layers[*4]. The translation from composite layers into layer transition rules proceeds as described in the following steps.

5.1 Translation from State Changes of Contexts into Those of Layers

First, it creates a parallel composition of independent context changes. **Figure 8** shows a parallel composition of the state tran-

[*4] In EventCJ, all the applicable layer transition rules are simultaneously applied when multiple events are declared on the same join point shadow [10], and/or when there are multiple layer transition rules on the same event [1].

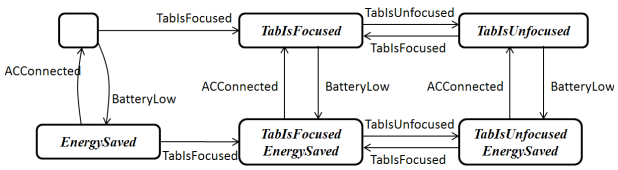

Fig. 8 Parallel-composed state transition diagram.

sition diagram shown in Fig. 1. In this composition, each state is a product of states of the transitions before composition.

Next, for each state, it assigns composite layers that become active at that state (in general, there can be several such composite layers). More precisely, it assigns composite layers whose `when` clauses become true at that state. In the following translation process, these composite layers are respectively converted into layers controlled by layer transition rules by removing the `when` clauses. Below, we call them layers when we do not have to distinguish them before and after the translation. We show the result of layer assignment for Fig. 8 as follows:

layer	node
`TabIsActive`	*TabIsFocused*
`TabIsInactive`	*TabIsUnfocused*
`TabIsInactive`	*EnergySaved*
`TabIsInactive`	*TabIsFocused,EnergySaved*
`TabIsInactive`	*TabIsUnfocused,EnergySaved*

Finally, it constructs the transitions of layers as follows. First, it merges states that are assigned the same layers into one state[*5]. Next, it constructs transitions as transition relations between corresponding "merged states." In this construction, it assigns the condition *unless* $L_1 \vee L_2 \vee \cdots \vee L_n$ (where $L_1 \cdots L_n$ are the names of contexts that are inactive at the source of the transition) to the labels of the transitions. These contexts are the set differences of the contexts, which can be active when the layer transition triggered by the specified event occurs from the contexts that can be active upon the occurrence of the context transition triggered by that event.

We formalize this merging process as follows. Let E be the set of events that appear in the program before the translation, let C be the set of atomic layers, and let L be the set of composite layers. The parallel-composed state transitions are defined as a quadruplet (St, R, st, l), where $St \subset \mathfrak{P}(C)$ is the set of states (where $\mathfrak{P}(C)$ is a powerset of C), $R \subset St \times St$ is the set of transition relations, $st \in St$ is the initial state, and $l : \mathfrak{P}(E) \to \mathfrak{P}(R)$ is the labeling function[*6]. The transitions of layers are defined as a quadruplet (St', R', st', l'), where $St' \subset \mathfrak{P}(L)$ is the set of states, $R' \subset St' \times St'$ is the set of transition relations, $st' \in St'$ is the initial state, and $l' : \mathfrak{P}(E) \times Gd \to \mathfrak{P}(R')$ is the labeling function (where $Gd \subset \mathfrak{P}(C)$ is a guard corresponding to the *unless* clauses in **Fig. 9**; i.e., if at least one of the contexts contained in Gd is active, the transition specified by l' does not occur). Let $layer : St \to St'$ be a function that performs the layer assignment

[*5] States with no assigned layers are merged into the initial state.

[*6] As mentioned above, in EventCJ, we can declare multiple events on the same join point shadow; in this case, multiple layer transition rules corresponding to those events are simultaneously applied. Thus, in general, transitions of contexts and layers are modeled as transitions between sets of layers (contexts), and each of them is labeled by a set of events.

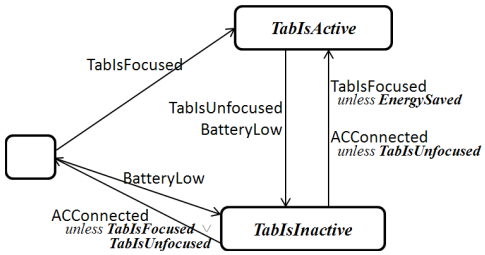

Fig. 9 State transition diagram of the merged layers.

```
1  St' := φ
2  R' := φ
3  l' := φ
4  for each e in 𝔓(E) {
5    Gd := φ
6    for each (s_i, s_j) in l(e)
7      Gd := Gd ∪ s_i
8    for each (s_i, s_j) in l(e) {
9      if (layer(s_i) ≠ layer(s_j)) {
10       St' := St' ∪ {layer(s_i)} ∪ {layer(s_j)}
11       R' := R' ∪ {(layer(s_i), layer(s_j))}
12       I := ⋃ t_k for all t_k ∈ equiv(s_i)
13       l' := l' ∪ {(e, (Gd − s_i) ∩ I) ↦ (layer(s_i), layer(s_j))}
14     }
15   }
16 }
```

Fig. 10 The algorithm for merging of states.

explained above. We define the function $equiv : St \to \mathfrak{P}(St)$, which calculates the equivalence classes for the equivalence relation $s \sim t \stackrel{\text{def}}{=} layer(s) = layer(t)$ on St as follows:

$$equiv(s) = \{t \in St | layer(s) = layer(t)\}$$

We show the algorithm for merging of states in **Fig. 10**. First, it obtains the set of contexts that can be activated when the context transition for the given event occurs (lines 6 and 7). Then, for each transition relation (s_i, s_j) for each event, it decides whether this transition still remains after the merging of states (line 9); if this transition remains, it populates St' and R' by adding corresponding elements (lines 10 and 11). Finally, it calculates the guard and adds it to the labeling function (lines 12 and 13).

The state transition diagram shown in Fig. 8 is merged into the state transition diagram shown in Fig. 9 (for simplicity, we also merge the edges whose source and target are the same into an edge where each label of the merged edges are listed). For example, `TabIsFocused` is a label of the transition from *TabIsInactive* to *TabIsActive*, which corresponds to the transition from *TabIsUnfocused* to *TabIsFocused* in Fig. 8; we have to distinguish it from two other `TabIsFocused`s labeled on the transitions between states placed at the lower part of the figure, which are merged into the same state with *TabIsUnfocused*. The former `TabIsFocused` occurs only when the system is not in *EnergySaved*; thus, we assign the condition *unless EnergySaved*.

If there is a large amount of states of context transitions, the number of states after the parallel composition explodes, because the latter is the product of the former. The parallel-composed state transitions are used only during the translation; thus, the number of states does not affect the runtime performance of the program. However, a very large number of states makes the translation time consuming. We can avoid this state explosion by con-

structing a parallel composition only for the state transitions of contexts that are referred from the respective composite layer, instead of performing parallel-composition of all state transitions. In other words, a composite layer depending on a large amount of contexts may make the translation time-consuming. Thus, a programmer should design the program not to over populate the `when` clause. For example, a composite layer that depends on a large amount of contexts may be constructed from partial methods, each of which depends only on a subset of those contexts. In this case, we may shrink the size of `when` clause by decomposing the layer.

5.2 Adding Events

The condition on each edge of the state transition diagram for layers indicates that, in addition to existing events, we need events that contain such condition. We can obtain such events by adding the `if` pointcut, inspecting whether the specified context (= atomic layer) is active to the existing events. The following event declaration is obtained from `TabIsFocused` by adding the information about $\neg$*EnergySaved*:

```
event TabIsFocused_1(ChangeEvent e)
 :after execution(void TabListener.stateChanged(*))
   &&args(e)
   &&if(!e.getSrc().getSelected().controller().
     lm.isActive(EnergySaved.ID))
 :sendTo(e.getSrc().getSelected().controller());
```

The symbol `lm` is a field that will be added by the EventCJ compiler to bind the object that manages the sequence of active layers. This object is equipped with the `isActive` method, which returns `true` when the layer identified by the identifier given by the parameter is active. By calling `isActive` with `ID` on the `EnergySaved` layer, the condition $\neg$*EnergySaved* is confirmed. Thus, we can declare `TabIsFocused_1` as an event that is generated only when `EnergySaved` is not active.

Using event declarations that have been generated in that manner, the state transition diagram shown in Fig. 9 is translated into the following layer transition rules:

```
1  transition TabIsFocused_1:
2    TabIsInactive ? TabIsInactive -> TabIsActive
3  | -> TabIsActive;
4
5  transition TabIsUnfocused:
6    TabIsActive ? TabIsActive -> TabIsInactive
7  | -> TabIsInactive;
8
9  transition BatteryLow:
10   TabIsActive ? TabIsActive -> TabIsInactive
11 | -> TabIsInactive;
12
13 transition ACConnected_1:
14   TabIsInactive ? TabIsInactive -> TabIsActive
15 | -> TabIsActive;
16
17 transition ACConnected_2:
18   TabIsInactive ? TabIsInactive ->;
```

Finally, composite layers are converted into atomic layers by removing the `when` clauses, which completes the translation from composite layers into atomic layers and layer transition rules. The translated code can be compiled by the existing EventCJ compiler.

6. Related Work

6.1 Non Layer-based COP Languages

Some COP languages do not use layers as units for the modularization of context-dependent variations of behavior. Subjective-C [8] is a COP language that extends Objective-C. Instead of declaring partial methods within a layer, in Subjective-C, we specify the context when the method is executable by adding the annotation `#context` to the method. A context is identified by its name, and it is explicitly switched within the program. Ambience [9] is a prototype-based COP language that changes executable code according to the context object that is implicitly given at the method calls. Lambic [19] is a COP language with the feature of conditional methods [7]. ContextErlang [18] is an extension of Erlang, which enables asynchronous context changes per-process.

These languages do not provide linguistic constructs for the control of layer activation such as `with`-blocks and layer transition rules. Instead, most of them provide the implicit switching of contexts according to the value of variables within the executing program, or imperative operations that are explicitly stated within the program to switch contexts (after the operation, the program definitely stays in that context unless the next operation to switch the context is triggered). Thus, in these languages, the problem described in Section 2 is unlikely to occur; however, it is difficult to model when, where, and how the context changes.

6.2 Implicit Activation of Layers

Costanza et al. proposed a method to analyze the dependency between layers using feature diagrams [5]. In this method, each feature corresponds to a layer. They also propose an extension of ContextL [6] with composite layers (that correspond to composite features). It shares some similarities with our approach; for example, it provides layer composition operators that are as expressive as compositions in feature diagrams (such as and-composition and or-composition), and it provides implicit layer activation. However, in Ref. [5], layers that are dependent on composite layers are implicitly activated, and the composite layers at the root of the composition have to be explicitly activated. On the other hand, in our approach, all atomic layers are explicitly activated by layer transition rules, and all composite layers are implicitly activated.

7. Conclusions

In this paper, we propose a composite layer, which is a new linguistic construct for COP languages, and use it to extend EventCJ. In the proposed method, only atomic layers are explicitly activated by layer transition rules. A composite layer declares a condition when it is active as a proposition where the names of layers are ground terms. It is active only when that condition is true. By carrying out small case studies, we confirmed that

the use of composite layers simplifies the program by reducing the number of required events and subrules compared with the case when we do not use them. With our approach, there are no layer transition rules where multiple context changes are tangled. Thus, our approach makes it possible to separate descriptions that are separated in the model of the real world within the program. Furthermore, we also show the steps for translating EventCJ with composite layers into (the original version of) EventCJ without composite layers. Therefore, in this paper, we solve the problems associated with the existing layer-based COP languages, i.e., we deal with the problem that the specification of which layer is activated and when this activation occurs becomes complex. Furthermore, we also address the case where the context changes that are separated in the real world are tangled within the program.

References

[1] Aotani, T., Kamina, T. and Masuhara, H.: Featherweight EventCJ: A core calculus for a context-oriented language with event-based per-instance layer transition, *COP'11* (2011).

[2] Appeltauer, M., Hirschfeld, R., Haupt, M. and Masuhara, H.: ContextJ: Context-oriented programming with Java, *Computer Software*, Vol.28, No.1, pp.272–292 (2011).

[3] Appeltauer, M., Hirschfeld, R. and Masuhara, H.: Improving the development of context-dependent Java application with ContextJ, *COP'09* (2009).

[4] Appeltauer, M., Hirschfeld, R., Masuhara, H., Haupt, M. and Kawauchi, K.: Event-specific software composition in context-oriented programming, *Proc. International Conference on Software Composition 2010* (*SC'10*), *LNCS*, Vol.6144, pp.50–65 (2010).

[5] Costanza, P. and D'Hondt, T.: Feature descriptions for context-oriented programming, *2nd International Workshop on Dynamic Software Product Lines* (*DSPL'08*) (2008).

[6] Costanza, P. and Hirschfeld, R.: Language constructs for context-oriented programming – an overview of ContextL, *Dynamic Language Symposium* (*DLS*) *'05*, pp.1–10 (2005).

[7] Ernst, M.D., Kaplan, C.S. and Chambers, C.: Predicate dispatch: A unified theory of dispatch, *ECOOP'98*, *LNCS*, Vol.1445, pp.186–211 (1998).

[8] González, S., Cardozo, M., Mens, K., Cádiz, A., Libbrecht, J.-C. and Goffaux, J.: Subjective-C: Bringing context to mobile platform programming, *SLE'11*, *LNCS*, Vol.6563, pp.246–265 (2011).

[9] González, S., Mens, K. and Cádiz, A.: Context-oriented programming with the ambient object systems, *Journal of Universal Computer Science*, Vol.14, No.20, pp.3307–3332 (2008).

[10] Hilsdale, E. and Hugunin, J.: Advice weaving in AspectJ, *AOSD'04*, pp.26–35 (2004).

[11] Hirschfeld, R., Costanza, P. and Nierstrasz, O.: Context-oriented programming, *Journal of Object Technology*, Vol.7, No.3, pp.125–151 (2008).

[12] Kamina, T., Aotani, T. and Masuhara, H.: EventCJ: A context-oriented programming language with declarative event-based context transition, *AOSD '11*, pp.253–264 (2011).

[13] Kamina, T., Aotani, T. and Masuhara, H.: Bridging real-world contexts and units of behavioral variations by composite layers, *Proc. 4th International Workshop on Context-Oriented Programming* (*COP'12*) (2012).

[14] Kamina, T., Aotani, T., Masuhara, H. and Tamai, T.: Context-Oriented Software Development with Use Cases, *Software Engineering Symposium 2011* (*SES2011*) (2011) (in Japanese).

[15] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Grisword, W.G.: An overview of AspectJ, *ECOOP'01*, pp.327–353 (2001).

[16] Lincke, J., Appeltauer, M., Steinert, B. and Hirschfeld, R.: An open implementation for context-oriented layer composition in ContextJS, *Science of Computer Programming*, Vol.76, No.12, pp.1194–1209 (2011).

[17] Salifu, M., Yu, Y. and Nuseibeh, B.: Specifying monitoring and switching problems in context, *RE'07*, pp.211–220 (2007).

[18] Salvaneschi, G., Ghezzi, C. and Pradella, M.: ContextErlang: Introducing context-oriented programming in the actor model, *AOSD'12* (2012).

[19] Vallejos, J., Costanza, P., Cutsem, T.V., De Meuter, W. and D'Hondt, T.: Reconciling generic functions with actors, *ACM SIGPLAN International Lisp Conference* (2009).

Tetsuo Kamina is an Assistant Professor at Graduate School of Education, the University of Tokyo. He received his B.A. from International Christian University in 1999, and his M.A. and Dr.A. (equivalent to Ph.D. in the Japanese system) from the University of Tokyo in 2002 and 2005, respectively. His research interests are programming languages and software engineering, in particular module mechanisms and their application to software development.

Tomoyuki Aotani is an Assistant Professor at the School of Information Science in Japan Advanced Institute of Science and Technology (JAIST). He received his B.Sc. from Hosei University in 2004 and M.A. and Ph.D. from the University of Tokyo in 2006 and 2009, respectively.

Hidehiko Masuhara is an Associate Professor at Graduate School of Arts and Sciences, the University of Tokyo. He received his B.S., M.S., and Doctorate in Computer Science from the University of Tokyo in 1992, 1994 and 1999 respectively. His research interests are programming languages and systems, in particular module mechanisms, execution techniques and development environments that help programmers to develop better software more efficiently.