Featherweight EventCJ: a Core Calculus for a Context-Oriented Language with Event-Based Per-Instance Layer Transition

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

Tetsuo Kamina Hidehiko Masuhara
University of Tokyo
{kamina,masuhara}@acm.org

Abstract

We propose Featherweight EventCJ, which is a small calculus for context-oriented languages with event-based per-instance layer controls like EventCJ. It extends ContextFJ with stores, labels and transitions for modeling the per-instance layer management, events and declarative layer transition rules, respectively.

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

General Terms Languages

Keywords Context-Oriented Programming, EventCJ, layer transition

1. Introduction

Context-oriented programming[6] is a technique to modularize context-dependent behavioral variations in a program, where those behavioral variations can be dynamically switched on and off in response to changes of execution contexts. Execution contexts include internal program states and external situation of the executing environment. For example, a navigation program running on a smartphone displays a map with different resolution (behavioral variations) depending on whether the device is outdoor and indoor (execution contexts).

EventCJ[10] is a Java-based context-oriented programming language that combines an existing modularization mechanism for context-dependent behavioral variations with a novel event-based layer activation mechanism. As for the modularization mechanism, we employ the layered partial methods that can be found in existing context-oriented programming languages such as ContextJ[6] and JCop[1]. As for the layer activation mechanism, EventCJ employs a mechanism based on events and transition rules.

The goal of the paper is to clarify the semantics of the newly introduced language mechanisms in EventCJ. While semantics for context-oriented languages have been formally discussed in the previous studies[2, 7, 13], the following features are unique to EventCJ and interesting enough to be formalized.

- Active layers are managed on a *per-instance basis*, rather than on a per calling-context basis.
- Changes of active layers are triggered by event declarations separately specified from the program, rather than a special construct (namely `with`) embedded in a program.
- Active layers are controlled by a set of transition rules that includes preconditions and sets of layers to be activated and deactivated.

We present a small calculus Featherweight EventCJ (FECJ in short) for two reasons. First, introduction of the events and the per-instance layer management requires clarification of semantics as they blur timings of layer changes. Second, a clear semantics is necessary to verify EventCJ programs, which is one of the important features of EventCJ. FECJ extends ContextFJ[7] with stores, labels and layer transition rules. Those three additional elements roughly correspond to the above three features that we want to clarify.

The rest of the paper is organized as follows. We first review the EventCJ's language mechanisms and reveal the semantics design issue in Section 2. Section 3 defines the syntax and an operational semantics for FECJ. We discuss related work in Section 4 and conclude the paper in Section 5 with some notes on future work.

2. EventCJ and Its Semantics Design Issue

Along with layers and partial methods available in most context-oriented languages, EventCJ offers two additional language constructs, namely event declarations and layer transition rules. Layers and partial methods are used to modularize context-dependent behaviors. Events and layer tran-

COP'11, July 25, 2011, Lancaster, UK.

```
class Navigation{
  void onGPSEnabled() {...}
  void onGPSDisabled() {...}
  void run() {}
  layer GPSNavi {
    void run() {
      proceed();
      /*show the position on the street map*/
    }
  }
  layer WifiNavi {
    void run() {
      proceed();
      /*show the position on the floor map*/
    }
  }
}
```

Figure 1. A class, layers and partial methods in a navigator

```
declare event GPSEvent(Navigation n)
  :before call(void Navigation.onGPSEnabled())
    &&target(n)
  :sendTo(n);
declare event WifiEvent(Navigation n)
  :before call(void Navigation.onGPSDisabled())
    &&target(n)
  :sendTo(n);
transition GPSEvent:
  WifiNavi ? WifiNavi -> GPSNavi |
  !GPSNavi ? GPSNavi;
transition WifiEvent:
  GPSNavi ? GPSNavi -> WifiNavi |
  !WifiNavi ? WifiNavi;
```

Figure 2. Events and layer transition rules in a navigator

sitions specify when and how contexts change. Unlike other context-oriented languages such as ContextJ[6] and JCop[1], EventCJ manages layers on a per-instance basis. In other words, each object has its own active layers in EventCJ.

This section first reviews the features by using a simplified pedestrian navigation system for a mobile device, and then shows the semantics design issue. A brief specification of the system is as follows. It uses either the global positioning system (GPS) or a wifi-based indoor positioning system (Wifi) to detect the current position, depending on whether the device is outdoor or indoor, respectively. We also assume, for simplicity, that the GPS is available if the device is outdoor, and that this navigation system uses the wireless LAN based positioning system only when the GPS is not available. The system then displays the current position on either a street map or a floor map, depending on the device's location. In other words, it displays a different kind of map depending on the current context, i.e., outdoor or indoor.

2.1 Layers and Partial Methods

As in other context-oriented languages, layers and partial methods in EventCJ are the mechanisms of modularizing context-dependent behaviors.

Figure 1 shows a simplified class, two layers and partial methods in the navigation program for Android. The class `Navigation` defines three methods; namely `onGPSEnabled` and `onGPSDisabled` called from the Android framework when the GPS becomes available and unavailable, respectively; and `run` called from the framework when the device's position is changed. It also defines two layers; `GPSNavi` (lines 5–10) defining the behavior in the outdoor context; and `WifiNavi` defining the behavior in the indoor context. A layer consists of partial methods (e.g., `run` at lines 6–9)[1]. `GPSNavi` and `WifiNavi` extend the original behavior of `run` by declaring around-type partial methods, either of which is executed instead of the original `run` method when the respective layer is active.

In around-type partial methods, we can call the special function `proceed`, which invokes the original method or another layer's around-type partial method. If, for example, `GPSNavi` and `WifiNavi` are both activated in this order on a `Navigation` object when `run` is executed, `run` in `GPSNavi` is first executed and then `run` in `WifiNavi` is executed upon `proceed`. When `run` in `WifiNavi` also calls `proceed`, the original `run` is finally executed.

2.2 Events and Layer Transition Rules

EventCJ manages active layers on each object. An event occurs to multiple objects before/after a certain operation is performed. The operations include a method call/execution, field set/get and object creation. Each transition rule is associated to the name of an event, and defines how the set of active layers of an object is changed.

Figure 2 shows the event declarations and layer transition rules for our navigation system. It defines two events, namely `GPSEvent` and `WifiEvent`, and transition rules associated to them. A `GPSEvent` (lines 1–4) is delivered to a `Navigation` object just before `onGPSEnabled` is called on the same object. The `sendTo` clause specifies to which object the event is delivered. Here we specify `n`, which binds the receiver object of the method call due to the `target` form. Similarly, a `WifiEvent` (lines 5–8) is delivered to a `Navigation` object just before `onGPSDisabled` is called on the same object.

Lines 9–14 are the layer transition rules[2]. The transition rule `Layer1 ? Layer1 -> Layer2` should be read as "if

[1] In addition to partial methods, a layer has `activate` and `deactivate` blocks. We don't explain them here because they are not important in this paper.

[2] We use a revised syntax from the original one[10].

Layer1 is active, then Layer1 is deactivated and Layer2 is activated".

Lines 9–11 define a layer transition rule upon a GPSEvent. It consists of two subrules. The first subrule declares that if the layer WifiNavi is active on the object that receives the event (i.e., a Navigation object), the layer is deactivated and GPSNavi is activated. The second subrule declares that if the layer GPSNavi is not active on the object, then GPSNavi is activated. The operator | concatenates these subrules. The subrule on the left hand side are checked first and the second subrule is not checked if the first subrule is applied. Similarly, lines 12–14 define the layer transition rule upon a WifiEvent.

2.3 Semantics Design Issue

How to define the semantics of layer transition rules in EventCJ is not trivial, since two or more rules can be applied at one time. Suppose, for example, there is an event E, two transition rules, namely L1?L1->L2 and L2?L2->L1 that are defined on E, and an object o on which the layer L1 is active. In such a case, we have to carefully define how the rules are evaluated when o receives E.

We face with a similar situation when two or more events are delivered to the same object at one time. Such a situation occurs when, for example, two events E1 and E2 are generated at the same time, and E1 is sent to the receiver of a method invocation and E2 is sent to the argument of a method invocation, which is actually the same object as the receiver, like o.m(o). Suppose the rule L1?L1->L2 is declared on E1 and the rule L2?L2->L1 is declared on E2. Again, we have to carefully define how these two rules are evaluated.

We can consider two candidates for these cases: evaluating just one rule that is selected in some way, and evaluating all of the rules. If we employ the former, it must be reasonably defined how to select one rule to be evaluated from all the rules. The latter also leads to another design issue about the evaluation order, that is, the rules should be evaluated parallely or sequentially.

EventCJ employs the latter semantics with the parallel evaluation strategy. It first evaluates every guard of layer transition rules and then deactivates layers. After the evaluation of guards and deactivation of layers, it activates layers. Hence, all the layers specified to be activated become active after the set of transitions. For instance, in the above examples, the guards of the two rules are firstly evaluated on the object o. Then L1 is deactivated and finally L2 is activated as specified by the rule L1?L1->L2. The another rule L2?L2->L1 is ignored, because o does not satisfy its guard that requires L2 must be active on o.

We believe that this semantics is more comprehensive and useful for the programmers than the another one, because it assures that the layers specified to be active in a rule always become active once their guards are satisfied no matter how many other rules are defined on the same event.

3. Featherweight EventCJ (FECJ)

This section gives a small calculus called Featherweight EventCJ, which models the EventCJ's layer (de)activation mechanism. It is built on top of ContextFJ[7], a pure functional calculus, yet is extended with stores in order to model the per-instance layer management. Furthermore, it employs deterministic call-by-value strategy to model the side-effect about layer transitions correctly.

FECJ models only *before* events on method calls and field accesses (i.e., events defined by using `before call(...)` and `before get(...)`) for simplicity. In addition, we assume that a method call or a field access matches at most one event declaration, and that every event declaration has exactly one `sendTo` clause with exactly one parameter.

We omit the definitions of typing rules because our extensions basically do not change the ones in ContextFJ.

3.1 Syntax

Let metavariables C, D, and E range over class names; L over layer names; f over field names; m over method names; ℓ over labels; v and w over addresses in stores; and x and y over variables, which include a special variable this. The abstract syntax of FECJ is as follows:

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

Overlines denote sequences, e.g., $\overline{\texttt{f}}$ stands for a possibly empty sequence $\texttt{f}_1, \cdots, \texttt{f}_n$ and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, and so on. The empty sequence is denoted by $\bullet$. We also abbreviate a sequence of pairs, writing "$\overline{\texttt{C f}}$" for "$\texttt{C}_1\ \texttt{f}_1, \cdots, \texttt{C}_n\ \texttt{f}_n$", where n is the length of $\overline{\texttt{C}}$ and $\overline{\texttt{f}}$, and similarly "$\overline{\texttt{C f}}$;" as shorthand for the sequence of declarations "$\texttt{C}_1\ \texttt{f}_1\texttt{;} \ldots \texttt{C}_n\ \texttt{f}_n\texttt{;}$" and "$\texttt{this.}\overline{\texttt{f=f}}\texttt{;}$" for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;} \ldots \texttt{;this.f}_n\texttt{=f}_n\texttt{;}$". We use commas and semicolons for concatenations. Sequences of field declarations, parameter names, layer names, and method declarations are assumed to contain no duplicate names. We also use a hat to denote an optional element, i.e., $\hat{\ell}$ denotes there is a label ℓ or no label. The empty optional is denoted by ϵ, which is usually omitted.

A class definition CL consists of its name, its superclass name, field declarations $\overline{\texttt{C f}}$, a constructor K, and method definitions $\overline{\texttt{M}}$. A constructor K is a trivial one that takes initial values of all fields and sets them to the corresponding fields. Unlike the examples in the last section, we do not

provide syntax for layers; partial methods are registered in a partial method table, explained below. A method M takes $\overline{\mathtt{x}}$ as arguments and returns the value of expression e. The method body consists of a single return statement and all constructs return values. An expression e can be a variable, field access, method invocation, object creation, proceed call, location, or a special form of method invocation $\mathtt{v<C,\overline{L},\overline{L}>.m(\overline{v})}$. A value is a location v. A layer transition rule t consists of four sequences of layer names. A partial store is a pair of a location and an object $\mathtt{new\ C(\overline{v})<\overline{L}>}$. A store is a sequence of partial stores, and used as a map from locations to objects. We write $dom(\mu)$ to denote all of the locations in the store, i.e., $dom(\mu) = \{\mathtt{v} | (\mathtt{v} \mapsto \mathtt{new\ C(\overline{w})<\overline{L}>}) \in \mu\}$.

A label attached to a field access or a method invocation denotes an event receiver. In other words, we simplify the event matching mechanism to labels attached to subexpressions that are designated as the receiver in the event declaration. For example, when GPSEvent in Figure 2 is declared for an expression e.onGPSDisabled() in EventCJ, the event is represented by adding a label to e in FECJ; i.e., rewriting the expression into $\mathtt{e}^{\ell}\mathtt{.onGPSDisabled()}$.

The number of the labels in each expression is at most one, which reflects the limitation that the sendTo clause takes exactly one parameter. For example, in the expression $\mathtt{e}^{\hat{\ell}_0}\mathtt{.m(}\overline{\mathtt{e}^{\hat{\ell}}}\mathtt{)}$, just one of $\hat{\ell}_0, \hat{\ell}_1, ..., \hat{\ell}_n$ is a label and others are empty. More formally, we assume that every expression e is well-formed, written as $wf(\mathtt{e})$, defined as follows:

$$\frac{}{wf(x)} \quad \text{(WF-VAR)}$$

$$\frac{wf(\mathtt{e})}{wf(\mathtt{e}^{\hat{\ell}}\mathtt{.f})} \quad \text{(WF-FIELD)}$$

$$\frac{\forall i \in \{0,..,n\}.wf(\mathtt{e}_i) \qquad \sum_{i=0}^{n} label(\mathtt{e}_i{}^{\hat{\ell}_i}) \leq 1}{wf(\mathtt{e}_0{}^{\hat{\ell}_0}\mathtt{.m(e}_1{}^{\hat{\ell}_1}\mathtt{,...,e}_n{}^{\hat{\ell}_n}\mathtt{)})} \quad \text{(WF-INVK)}$$

$$\frac{\forall i \in \{1,...,n\}.wf(\mathtt{e}_i)}{wf(\mathtt{new\ C(e}_1\mathtt{,...,e}_n\mathtt{)<\overline{L}>})} \quad \text{(WF-NEW)}$$

$$\frac{\forall i \in \{1,...,n\}.wf(\mathtt{e}_i)}{wf(\mathtt{proceed(e}_1\mathtt{,...,e}_n\mathtt{)})} \quad \text{(WF-PROCEED)}$$

where $label(\mathtt{e}^{\hat{\ell}})$ is 1 when $\hat{\ell}$ is a label, and 0 otherwise. We omit the rules for locations and special form of method invocation because they should not appear in the source code.

The expressions v and $\mathtt{v<D,\overline{L}',\overline{L}>.m(\overline{w})}$, where $\overline{\mathtt{L}}'$ is assumed to be a prefix of $\overline{\mathtt{L}}$, are special run-time expressions and not supposed to appear in either classes, partial methods or the main expression. $\mathtt{v<D,\overline{L}',\overline{L}>.m(\overline{w})}$ means that m is going to be invoked on v. The annotation $\mathtt{<D,\overline{L}',\overline{L}>}$ indicates where method lookup should start when a method and proceed is invoked. More concretely, $\mathtt{<D,(L}_1\mathtt{;\cdots;L}_i\mathtt{),(L}_1\mathtt{;\cdots;L}_n\mathtt{)>}$ $(i \leq n)$ means that the search for the method definition will start from class D in layer $\mathtt{L}_i$. For example, the usual method invocation $\mathtt{v.m(\overline{w})}$, whose receiver and arguments are already reduced to values, is semantically equivalent to $\mathtt{v<C,\overline{L},\overline{L}>.m(\overline{w})}$, where $\mu(\mathtt{v}) = \mathtt{new\ C(\overline{v})<\overline{L}>}$ for some store μ. The triple plays the role of a cursor in the method lookup procedure and the behavior of proceed as in ContextFJ.

A transition $\overline{\mathtt{L}}_1 : \overline{\mathtt{L}}_2 ? \overline{\mathtt{L}}_3 \rightarrow \overline{\mathtt{L}}_4$ should be read as "if all of the layers in $\overline{\mathtt{L}}_1$ are active and none of the layers in $\overline{\mathtt{L}}_2$ is active, deactivate $\overline{\mathtt{L}}_3$ and activate $\overline{\mathtt{L}}_4$". For example, the first subrule for GPSEvent(Figure 2) is written as

$$\mathtt{WifiNavi\ :\ \bullet\ ?\ WifiNavi} \rightarrow \mathtt{GPSNavi}$$

and the second subrule is written as

$$\mathtt{\bullet\ :\ WifiNavi;GPSNavi\ ?\ \bullet} \rightarrow \mathtt{GPSNavi}$$

The second sequence contains WifiNavi to encode the behavior of the or-operator.

An FECJ program $(CT, PT, TT, \mathtt{e})$ consists of a class table *CT* that maps a class name to a class definition, a partial method table *PT* 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, and a well-formed expression e that corresponds to the body of the main method. In the paper, we assume *CT*, *PT* and *TT* to be fixed and satisfy the following sanity conditions:

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

Lookup functions. FECJ employs the same auxiliary functions as ContextFJ to look up field and method definitions. They are defined by the rules in Figure 3. The function $fields(\mathtt{C})$ returns a sequence $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ of pairs of a field name and its type by collecting all field declarations from C and its superclasses. The function $mbody(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}_1, \overline{\mathtt{L}}_2)$ returns the parameters and body $\overline{\mathtt{x}}.\mathtt{e}$ of method m in class C when the search starts from $\overline{\mathtt{L}}_1$; the other layer names $\overline{\mathtt{L}}_2$ keep track of the layers that are activated when the search initially started. It also returns the information on where the method has been found—the information will be used in reduction rules to deal with proceed. As we mentioned already, the method

$\boxed{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$

$$\mathit{fields}(\mathtt{Object}) = \bullet \qquad \text{(F-OBJECT)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \ldots\ \} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}'}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}', \overline{\mathtt{C}}\ \overline{\mathtt{f}}} \qquad \text{(F-CLASS)}$$

$\boxed{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{D}, \overline{\mathtt{L}}''}$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \ldots\ \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}\ \ldots\}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{C}, \bullet} \qquad \text{(MB-CLASS)}$$

$$\frac{\mathit{PT}(\mathtt{m}, \mathtt{C}, \mathtt{L}_0) = \mathtt{C}_0\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e;}\ \}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, (\overline{\mathtt{L}}'; \mathtt{L}_0), \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{C}, (\overline{\mathtt{L}}'; \mathtt{L}_0)} \qquad \text{(MB-LAYER)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ \ldots\ \overline{\mathtt{M}}\ \} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{L}}, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{E}, \overline{\mathtt{L}}'}{\mathit{mbody}(\mathtt{m}, \mathtt{C}, \bullet, \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{E}, \overline{\mathtt{L}}'} \qquad \text{(MB-SUPER)}$$

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

Figure 3. Lookup functions.

definition is searched for in class C in all activated layers and the base definition and, if there is none, then the search continues to C's superclass. By reading the rules in a bottom-up manner, we can read off the recursive search procedure. The first rule of *mbody* means that m is found in the base class definition C (notice the third argument is $\bullet$) and the second that m is found in layer $\mathtt{L}_0$. The third rule, which deals with the situation where m is not found in a base class (expressed by the condition $\mathtt{m} \notin \overline{\mathtt{M}}$), motivates the fourth argument of *mbody*. The search goes on to C's superclass D and has to take all activated layers into account; so, $\overline{\mathtt{L}}$ is copied to the third argument in the premise. The fourth rule means that, if C of $\mathtt{L}_0$ does not have m, then the search goes on to the next layer (in $\overline{\mathtt{L}}'$) leaving the class name unchanged.

3.2 Operational Semantics

The operational semantics of FECJ is given by a reduction relation of the form $\mathtt{e}|\mu \longrightarrow \mathtt{e}'|\mu'$, read "expression e under the store μ reduces to e′ under μ'". Here, μ and μ' must not contain duplicate names.

The main rules are shown in Figure 4. Note that FECJ employs call-by-value strategy so that method invocation and field access are evaluated only when the receivers and arguments are reduced to values. We write $\hat{\mathtt{p}}\mu$ to denote

$\boxed{\mathtt{e}|\mu \longrightarrow \mathtt{e}'|\mu'}$

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

$$\frac{\mu(\mathtt{v}) = \mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{)<}\overline{\mathtt{L}}\mathtt{>} \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\mathtt{v.f}_i|\mu \longrightarrow \mathtt{w}_i|\mu} \qquad \text{(R-FIELD)}$$

$$\frac{\mu(\mathtt{v}_0) = \mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{)<}\overline{\mathtt{L}}\mathtt{>} \qquad \mathtt{v}_0\mathtt{<C,}\overline{\mathtt{L}}\mathtt{,}\overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{v}}\mathtt{)}|\mu \longrightarrow \mathtt{e}|\mu'}{\mathtt{v}_0\mathtt{.m(}\overline{\mathtt{v}}\mathtt{)}|\mu \longrightarrow \mathtt{e}|\mu'} \qquad \text{(R-INVK)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{C}'', \bullet}{\mathtt{v<C}', \overline{\mathtt{L}}', \overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)}|\mu \longrightarrow \left[\begin{array}{l}\mathtt{v/this},\\ \overline{\mathtt{w}}/\overline{\mathtt{x}}\end{array}\right]\mathtt{e}|\mu} \qquad \text{(R-INVKB)}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}', \overline{\mathtt{L}}', \overline{\mathtt{L}}) = \overline{\mathtt{x}}.\mathtt{e}\ \mathtt{in}\ \mathtt{C}'', (\overline{\mathtt{L}}''; \mathtt{L}_0)}{\mathtt{v<C}', \overline{\mathtt{L}}', \overline{\mathtt{L}}\mathtt{>.m(}\overline{\mathtt{w}}\mathtt{)}|\mu \longrightarrow \left[\begin{array}{ll}\mathtt{v} & \mathtt{/this},\\ \overline{\mathtt{w}} & /\overline{\mathtt{x}},\\ \mathtt{v<C}'', \overline{\mathtt{L}}'', \overline{\mathtt{L}}\mathtt{>.m/proceed}\end{array}\right]\mathtt{e}|\mu} \qquad \text{(R-INVKP)}$$

$$\frac{\mathtt{w} \notin \mathit{dom}(\mu)}{\mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)}|\mu \longrightarrow \mathtt{w}|(\mu, \mathtt{w} \mapsto \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{)<}\bullet\mathtt{>})} \qquad \text{(R-NEW)}$$

$$\frac{\mathtt{e}|\mu \longrightarrow \mathtt{e}'|\mu'}{G[\mathtt{e}^{\hat{\ell}}]|\mu \longrightarrow G[\mathtt{e}'^{\hat{\ell}}]|\mu'} \qquad \text{(RC-LABEL)}$$

Figure 4. Featherweight EventCJ: Reduction rules.

updating the store μ; $(\mathtt{v} \mapsto \mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{)<}\overline{\mathtt{L}}\mathtt{>})\mu = \mu'$ where μ' satisfies the following three conditions, $\mathit{dom}(\mu) = \mathit{dom}(\mu')$, $\mu(\mathtt{w}) = \mu'(\mathtt{w})$ for all $\mathtt{w} \in \mathit{dom}(\mu) \setminus \{\mathtt{v}\}$, and $\mathit{dom}(\mu') = \mathtt{new\ C(}\overline{\mathtt{w}}\mathtt{)<}\overline{\mathtt{L}}\mathtt{>}$ if $\mathtt{v} \in \mathit{dom}(\mu)$.

The first rule R-LABEL is for applying transitions. The form $J[\cdot]$ represents evaluation contexts for operations defined as follows:

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

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 v, which reflects the fact that each expression is well-formed. The function *transit*, which is explained later, applies the transitions for the label $\hat{\ell}$ to the object at the location v in the store μ and tells how the store is updated.

The second rule R-FIELD is for field access. The function *fields* tells which argument to `new C(..)`, which is obtained by looking up the store, corresponds to $\mathtt{f}_i$.

$\boxed{\mathit{transit}(\mu, \mathtt{v}, \ell) = \mathtt{p}}$

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

(TR-TRANSITION)

$\boxed{\mathit{applicable}(\mathtt{t}, \overline{\mathtt{L}})}$

$$\frac{\overline{\mathtt{L}}_1 \cap \overline{\mathtt{L}} = \overline{\mathtt{L}}_1 \qquad \overline{\mathtt{L}}_2 \cap \overline{\mathtt{L}} = \bullet}{\mathit{applicable}(\overline{\mathtt{L}}_1 : \overline{\mathtt{L}}_2 ? \overline{\mathtt{L}}_3 \rightarrow \overline{\mathtt{L}}_4, \overline{\mathtt{L}})}$$

(PRED-APPLICABLE)

Figure 5. Transition rules

The next three rules are for method invocation. R-INVK is for method invocation where the cursor of the method lookup procedure has not been initialized; the cursor is set to be at the receiver's class and the currently activated layers.

In R-INVKB, v is the receiver and <C′,$\overline{\mathtt{L}}'$,$\overline{\mathtt{L}}$> is the location of the cursor. When the method body is found in the base-layer class C′′ (denoted by "in C′′, •"), the whole expression reduces to the method body where the formal parameters $\overline{\mathtt{x}}$ and this are replaced with the actual arguments $\overline{\mathtt{w}}$ and the receiver v, respectively. The store is not updated because it merely looks up the method body and substitutes the variables. R-INVKP deals with the case where the method body is found in layer $\mathtt{L}_0$ in class C′′. In this case, proceed in the method body is replaced with the invocation of the same method, where the receiver's cursor points to the next layer $\overline{\mathtt{L}}''$ (dropping $\mathtt{L}_0$). Since the meaning of the annotated invocation is not affected by the layers in the context (note that $\overline{\mathtt{L}}'''$ are not significant in these rules), the substitution for proceed also means that their meaning is the same throughout a given method body. The store is not updated because of the same reason to R-INVKB.

R-NEW extends the current store μ by simply adding a new partial store to it. The new partial store consists of a fresh location w and the object with no active layer.

RC-LABEL reduces subexpressions with optional labels. $G[\cdot]$ forms evaluation contexts, defined as follows:

$$G ::= [\,] \mid G.\mathtt{m}(\overline{\mathtt{e}}^{\overline{\hat{\ell}}}) \mid \mathtt{v}^{\hat{\ell}}.\mathtt{m}(\overline{\mathtt{w}}^{\overline{\hat{\ell}}}, G, \overline{\mathtt{e}}^{\overline{\hat{\ell}}'}) \mid G.\mathtt{f} \mid \mathtt{new\ C(}\overline{\mathtt{v}}\mathtt{,}G\mathtt{,}\overline{\mathtt{e}}\mathtt{)}$$

We write $G[\mathtt{e}^{\hat{\ell}}]$ for the ordinary expression obtained by replacing the hole in G with $\mathtt{e}^{\hat{\ell}}$. Unlike FJ nor ContextFJ, FECJ requires the receiver and all of the arguments on the left of the redex to be values. This naive reductions are necessary because FECJ has side-effects, i.e., the choice whether the receiver or one of the arguments are reduced first can affect the result of a method invocation.

Transitions. The function *transit* basically changes the set of active layers on an object by applying transitions. Figure 5 shows the definition. We use set-like notations to denote the operations on sequences with preserving the order. $\{\mathtt{t} \in \mathtt{t}_1\mathtt{,..,t}_n | \mathit{pred}(t)\}$ is the sequence $\mathtt{t}_1\mathtt{'},\mathtt{..,t}_k\mathtt{'}$ where $\mathtt{t}_1\mathtt{'}$ and $\mathtt{t}_k\mathtt{'}$ are the first and last element in $\mathtt{t}_1\mathtt{,...,t}_n$ that satisfies the predicate *pred*, respectively, and for all $\mathtt{t}_i$ and $\mathtt{t}_j$ in $\mathtt{t}_1\mathtt{',..,t}_k\mathtt{'}$ and $i < j$ there exists $\mathtt{t}_{i'}$ 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 $i' < j'$. $\overline{\mathtt{L}} \setminus \overline{\mathtt{L}}'$ means the operation that removes all the elements in the sequence $\overline{\mathtt{L}}'$ from $\overline{\mathtt{L}}$ without changing the order of $\overline{\mathtt{L}}$. Here we ignore the order in $\overline{\mathtt{L}}'$. $\overline{\mathtt{L}} \oplus \overline{\mathtt{L}}'$ concatenates the two sequences without any duplications, i.e., $(\mathtt{L}_1\mathtt{;..;L}_i\mathtt{;..;L}_n) \oplus (\mathtt{L}_1\mathtt{';..;L}_i\mathtt{';L}_i\mathtt{;L}_{i+2}\mathtt{';..;L}_k\mathtt{'})$ is the sequence $\mathtt{L}_1\mathtt{;..;L}_i\mathtt{;..;L}_n\mathtt{;L}_1\mathtt{';..;L}_i\mathtt{';L}_{i+2}\mathtt{';..;L}_k\mathtt{'}$. We also write $\oplus\{\overline{\mathtt{L}}, .., \overline{\mathtt{L}}'\}$ for the operation that appends all of the sequences of layers, i.e., it means $\overline{\mathtt{L}} \oplus \mathtt{..} \oplus \overline{\mathtt{L}}'$. $\overline{\mathtt{L}} \cap \overline{\mathtt{L}}'$ removes all of elements in $\overline{\mathtt{L}}'$ that are not in $\overline{\mathtt{L}}$. The result is a sequence in which all the elements are ordered as in $\overline{\mathtt{L}}$.

In TR-TRANSITION, the function *transit* takes a store μ, location v and label ℓ and returns a partial store consists of the location v and the object with active layers affected by the transition rules for ℓ. The transitions are obtained by looking up the transition table *TT*. The predicate $\mathit{applicable}(\overline{\mathtt{L}}_1 : \overline{\mathtt{L}}_2 ? \overline{\mathtt{L}}_3 \rightarrow \overline{\mathtt{L}}_4, \overline{\mathtt{L}})$ means that the transition t satisfies its precondition, i.e., all of the layers in $\overline{\mathtt{L}}_1$ is in $\overline{\mathtt{L}}$ and none of the layers in $\overline{\mathtt{L}}_2$ is in $\overline{\mathtt{L}}$, as defined by PRED-APPLICABLE. Layers to be deactivated and activated are computed on the transitions. In other words, all of the transitions on the same label are applied at the same time.

3.3 Example: Evaluation of Layer Transitions

This section shows how the multiple layer transition rules are evaluated by using a simple example. When there are two or more layer transition rules on the delivered event(s), the reduction proceeds in our semantics as follows; firstly, it filters out every rule in which its guard is not satisfied; secondly, it deactivates all of the layers specified to be deactivated in the filtered rules; finally, it activates all of the layers specified to be activated in the filtered rules.

Suppose, for example, $\mathit{TT}(\ell_1)$ has two transition rules, L1: • ?L1→L3 and L2: • ?L2→L1. In this setting, the expression $\mathtt{v}^{\ell_1}$.m() under v ↦ new C()<L1> is reduced to v.m() under v ↦ new C()<L2> by R-LABEL because only the first transition rule is applicable. When the second transition rule is L1: • ?L1→L3, the store environment becomes v ↦ new C()<L2,L3>. In this case, both rules are applicable, and thus L1 is deactivated and L2 and L3 are activated.

4. Related Work

The event declarations and layer transition rules in EventCJ are similar to pointcuts and advice in aspect-oriented programming languages such as AspectJ[11]. Walker et al. presented the idea of annotating expressions with labels and us-

ing a label to identify a join point (shadow) in [14]. A piece of advice is represented as a pair of a label and function and triggered when control flow reaches a point annotated with the label, similar to FECJ. MiniMao$_1$[3], AFGJ[9] and StrongAspectJ[4] are calculi for aspect-oriented languages based on Java-like object-oriented languages. They model not only the effects of advice but also pointcut matchings. The ideas might be helpful to extend FECJ so that it models how the event declarations are evaluated.

Since FECJ formalizes some aspects of event handling in EventCJ, it is also worth to compare it to other formal studies on event-based programming. FEJ [5] is a core language of EventJava [5, 8]. In FEJ, an event is an asynchronous method that returns no values; i.e., we can invoke an event as a normal method call but the caller does not wait for the return of that call. Ptolemy [12] is another example of event-based programming language whose semantics is formally given. Ptolemy models an event as a closure that is executed on behalf of execution of corresponding event handler. Unlike those languages, FECJ models an event as a join-point, and provides a novel feature of combining events with (de)activation of layers.

5. Conclusions

We developed a small calculus called FECJ in order to formalize the semantics of EventCJ. We extended ContextFJ with stores, labels and layer transition rules so as to express unique features in EventCJ including per-instance layer management, event-based triggering of layer transitions, and rule-based layer transition control.

One of our future work is to formally discuss verification processes in EventCJ, in particular the SPIN-based verification with semi-automated generation of Promela code [10].

References

[1] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, and K. Kawauchi. Event-specific software composition in context-oriented programming. In *SC '10*, pages 50–65, 2010.

[2] D. Clarke and I. Sergey. A semantics for context-oriented programming with layers. In *COP '09*, pages 1–6, 2009.

[3] C. Clifton and G. T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.

[4] B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: Flexible and safe pointcut/advice bindings. In *AOSD '08*, pages 60–71, 2008.

[5] P. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *ECOOP 2009*, pages 570–594, 2009.

[6] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3): 125–151, 2008.

[7] R. Hirschfeld, A. Igarashi, and H. Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *FOAL '11*, pages 19–23, 2011.

[8] A. Holzer, L. Ziarek, K. R. Jayaram, and P. Eugster. Putting events in context: aspects for event-based distributed programming. In *AOSD'11*, pages 241–252, 2011.

[9] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63 (3):267–296, 2006.

[10] T. Kamina, T. Aotani, and H. Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD '11*, pages 253–264, 2011.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.

[12] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP 2008*, volume 5142 of *LNCS*, pages 155–179, 2008.

[13] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *OOPSLA '08*, pages 525–542, 2008.

[14] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP '03*, pages 127–139, 2003.