EventCJ: A Context-Oriented Programming Language with Declarative Event-based Context Transition[*]

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

This paper proposes EventCJ, a context-oriented programming (COP) language that can modularly control layer activation based on user-defined events. In addition to defining context-specific behaviors by using existing COP constructs, the EventCJ programmer declares *events* to specify when and on which instance layer switching should happen, and *layer transition rules* to specify which layers should be activated/deactivated upon events. These constructs enable controlling layer activation on a per-instance basis, separately from a base program. We also demonstrate an approach to verify safety properties of layer transitions by using a model checker. With these advantages, EventCJ enables more modular descriptions of context-aware programs, especially when layer switching is triggered in many places of a program, or by activities external to the base program. We implemented a prototype EventCJ compiler with Eclipse IDE support.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Context-oriented programming, Instance-specific layer activation, Verification

1. INTRODUCTION

Context awareness is a key requirement in many application domains such as mobile computing and adaptive user interfaces. Programs in such a domain have many functions that behave slightly differently depending on the current context, including system configuration, user preference, course of past operations, and conditions of computing environment. Context-oriented programming (COP) [20] allows programmers to implement dynamically switching behaviors in a modular manner by providing the following linguistic mechanisms, namely, a *layer*, which is an abstraction of context-dependent behaviors, a *partial method* in a layer that defines a behavior specific to a particular context, and *layer activation* that dynamically redirects a method call to a partial method in an active layer. For example, a pedestrian navigation system on a mobile device switches positioning methods, such as GPS, wireless LAN, and RFID tags, and changes map views depending on the current location. In a COP language, methods that control a map view can be implemented as partial methods in the "indoor" and "outdoor" contexts, one of which is activated according to the current location of the device. There are many implementations of COP languages to date [3, 12, 14, 19, 28, 31].

One of the design issues in COP languages is the means of controlling layer activation, namely, *when and which* layers should be activated and deactivated, and *at which parts of program execution* that layer activation and deactivation should take effect. Most existing COP languages provide block-structured constructs, such as a `with` statement in ContextJ [20] and JCop [5], with the assumption that context changes trigger the execution of context-dependent behavior. With this assumption, the block-structure syntactically ensures the deactivation of activated layers when execution escapes from the block.

Unlike existing COP languages, we aim to support systems where changes of contexts and executions of context-dependent behaviors happen at different points in a program text and/or in different threads of execution. As pointed out by Appeltauer et al. [4] and Desmet et al. [15], it is not easy for the block-structured constructs to support this type of programs. In the above-mentioned pedestrian navigation system, a change of context (i.e., a change of the device's environment) is either notified by a call-back method from a framework, or detected by a thread that polls the values of positioning sensors of the device. Such notification or detection happens concurrently with context-dependent behaviors, i.e., the execution of a method that displays a map.

In this paper, we propose a COP language called EventCJ that can separate the control of layer activation and the exe-

[*]This paper is an extended version of a preliminary workshop paper [22].

cution of context-dependent behavior, instead of providing a block-structured layer activation construct. The separation is achieved and compensated by the following mechanisms and approaches.

- EventCJ manages active layers based on *events* that trigger *transition of layers*, instead of controlling the layer activation on the basis of the dynamic scope of executions. Events are specified using an AspectJ's pointcut-like construct. Transitions are specified by a rule-based sub-language.
- EventCJ manages active layers on a *per-instance* basis, instead of on a per-thread basis as other COP languages do. This is because layer activation in EventCJ is no longer tightly coupled with the dynamic scope of execution, which makes per-thread management inappropriate.
- We demonstrate an approach to verify safety properties of layer transitions by using the SPIN model checker [21]. Although this approach requires manual specification of the base program's behavior, it compensates for the loss of disciplined layer activation enforced by the block-structured layer activation construct.

Note that the above mechanisms are used merely for controlling layer activation. To describe context-dependent behaviors, EventCJ uses basically the same syntax and method dispatching strategy for the layers and the partial methods as used in ContextJ [20] and JCop [5].

To study EventCJ's feasibility and applicability to real-world problems, we implemented a prototype compiler, conducted a preliminary performance evaluation, and carried out a case study implementing a program editor [4] and a pedestrian navigation system in EventCJ.

The rest of the paper is organized as follows. Section 2 shows a motivating example of a navigation system running on a mobile device that demonstrates that the existing COP features cannot easily support a certain kind of context management. Section 3 presents the design of EventCJ. We discuss an approach to verify safety properties by using a model checker in Section 4. Section 5 describes the implementation of EventCJ with its performance measurements. Section 6 presents a case study that investigates feasibility of EventCJ's language features in practical application programs. Section 7 discusses related work. Section 8 concludes the paper.

2. MOTIVATION

2.1 Pedestrian Navigation System

This section describes the development of event-based context transition mechanisms by using a simplified pedestrian navigation system running on a mobile device. The system uses either the global positioning system (GPS) or the wireless LAN based positioning system to detect the current position, depending on whether the device is outdoors or inside a building, respectively, and displays the current position of the device. The example is developed on top of the Android SDK[1], which provides APIs for displaying street maps and accessing the resources of the mobile device.

[1]`http://developer.android.com/sdk/`

```
class Navigation extends MapActivity
    implements Runnable, LocationListener {
  MapView mapView;
  MyLocationOverlay overlay;
  BuildingGuide buildingGuide;

  void onStatusChanged(...) {...}
  void run() {}

  void onCreate(Bundle status) {
    ... overlay.runOnFirstFix(this); ...
  }

  layer GPSNavi{
    after void onStatusChanged(...) {...}
    after void run() {
      Location loc = overlay.getMyLocation();
      mapView.getController().animateTo(loc);
    }
  }
  layer WifiNavi{
    after void onStatusChanged(...) {...}
    after void run() {
      Location loc = overlay.getMyLocation();
      buildingGuide.updateFloorPlan(loc);
    }
  }
}
```

Figure 1: Skeleton of pedestrian navigation system with layers and partial methods.

The implementation of the system has two context-dependent behaviors. First, it displays the current position on either a street map or a floor map, depending on the device's location. Second, when the device is on an airplane, the system uses neither positioning systems. In the followings, we discuss only the former context-dependent behavior.

In this paper, we assume for simplicity that the GPS is available iff the device is outdoors, and that this navigation system uses the wireless LAN based positioning systems only when the GPS is not available. In the followings, we call outdoors as the GPS-specific context, and indoors as the wireless LAN based positioning system specific context.

With an existing COP language, we can modularize these context-dependent behaviors by using partial methods and layers. Figure 1 shows a skeleton of the navigation system implementing the former context-dependent behavior in ContextJ [3]. The instance variables `mapView` and `overlay` are provided from the Android framework API. They respectively display a street map by using Google Maps API internally, and obtain the device's geographical location by using either the GPS or the wireless LAN signals. The instance variable `buildingGuide` is a navigation system's own component that displays a floor plan in a building by using a database stored in the device.

Following the method and instance variable declarations that are not context-specific, we declare the GPS navigation specific behaviors in the `GPSNavi` layer and the wireless LAN navigation specific behaviors in the `WifiNavi` layer.

If the `GPSNavi` layer is active, a call to the `run` scrolls the street map using animation, which is executed on a different thread, to update its current position when `overlay` detects a new position. If the `WifiNavi` layer is active, a call to the `run` updates the floor plan by calling `updateFloorPlan` when `overlay` detects a new position.

2.2 Problem of Controlling Layer Activation

If the application program runs each of behavioral variation at particular points in the program text, the `with` statement in ContextJ is useful to activate corresponding layers. For example, a method that demonstrates the navigation system in a building can be defined by using the `with` statement.

```
void runWifiDemo(Navigation n) {
  with (WifiNavi) { while (true) n.run(); }
}
```

2.2.1 *Cases Where* `with` *Cannot be Applied*

The `with` statement in ContextJ is, however, not suitable to control context-specific behaviors whose layer activation is triggered by other parts in the program, or by other threads of execution. For example, in the pedestrian navigation system, activation of the `WifiNavi` layer may be triggered by the following event handler `onStatusChanged` that handles an event detected by a change of the GPS system status indicating the entrance of a building:

```
void onStatusChanged(...) {
  overlay.onProviderDisabled("gps");
  wifiManager.setWifiEnabled(true);
  // WifiNavi shall be active, but...
}
```

However, the `onStatusChanged` method itself does not call the `run` method, which should exhibit context-dependent behavior. Furthermore, the thread executing this event handler might be different from that executing the `run` method; however, ContextJ can activate layers only under a particular control flow.

2.2.2 *Workaround and its Problem*

A workaround in ContextJ is to use first-class layers, i.e., storing effective layers in a variable, and activating the stored layers whenever a context-specific behavior is needed. In the pedestrian navigation system, this can be done by declaring a global variable `currentLayers`, letting `onStatusChanged` update the variable, and inserting a `with` statement into *all* calls to `run` as below:

```
with(Navigation.currentLayers){ run(); }
```

This workaround easily faces with the scattering problem because we have to update the current layers in many places in the program text, as well as to wrap all of the invocations of methods that have context-dependent behavior. Furthermore, the `run` method can be called from libraries that we cannot insert such a statement.

A possible solution to the scattering problems is to use an AspectJ-like pointcut and low-level layer activation/deactivation primitives (such as the ones provided from the ContextJ reflection API) to declare a piece of advice that makes layer switching.

```
pointcut WifiEvent(): execution(
  void Navigation.onStatusChanged);
after(Navigation n): WifiEvent() && this(n)
       && if(Layer.isActive(GPSNavi)){
  overlay.onProviderDisabled("gps");
  Composition.deactivateLayer(GPSNavi);
  Composition.activeLayer(WifiNavi);
  n.overlay.onProviderEnabled("network");
  n.wifiManager.setWifiEnabled(true);
}
```

Figure 2: A workaround that uses low-level layer activation primitives in AspectJ advice.

Figure 2 shows the pedestrian navigation example written in a hypothetical language in which we can use both ContextJ's layer activation primitives and AspectJ's advice. The `WifiEvent` pointcut captures the execution of the `onStatusChanged` method declared in `Navigation`. The advice is executed after the `onStatusChanged` method is executed. It first checks activation of the `GPSNavi` layer by using the `isActive` method provided by the ContextJ reflection API. If the layer is active, it turns the GPS receiver off, deactivates the `GPSNavi` layer, activates the `WifiNavi` layer, and then starts the wireless LAN sensor.

However, these imperative primitives for controlling layer activation make it difficult to reason about safety properties in terms of active layers, such that the`GPSNavi` and `WifiNavi` layers should never be active at the same time. With a block-structured layer activation construct, it is not difficult to reason about this property, because the construct guarantees that the activated layer is inactive after finished its execution. On the other hand, ensuring this property is quite difficult for the programs written with the layer activation primitives.

3. DESIGN OF EVENTCJ

This section proposes a new programming language called EventCJ that addresses the aforementioned problems. First, unlike the `with` statement, EventCJ separates the code for layer activation management and context-dependent behaviors, and provides a way to declaratively specify rules for switching layers. Second, instead of activating layers in dynamic scope of a thread execution, EventCJ activates and deactivates layers at different points in a program on a *per-instance* basis. Third, to compensate for the lack of block structure, we propose an approach to verify properties of layer activation by using the SPIN model checker.

EventCJ provides, in addition to the layer declaration constructs that are borrowed from ContextJ, new language constructs, namely, *event declarations* and *layer transition rules*. An event declaration defines an event that will trigger layer transitions. An event is specified by selecting the join points of the base program. A layer transition rule specifies target instances whose layer activation will change, application conditions of the rule, and layers to be (de-)activated. Figure 3 illustrates the relationship among a base program, layer declarations, event declarations and layer transition rules.

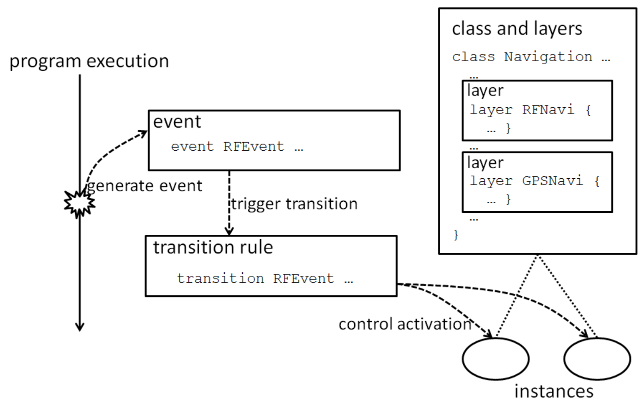

Figure 3: Relationship among a base program, layer declarations, event declarations and layer transition rules.

3.1 Layer Declarations

Syntax of layer declarations in EventCJ is as follows:

LayerDecl	::=	`layer` *ID* `{` *LayerMemDecl*$*$ `}`
LayerMemDecl	::=	*PartialMethDecl* | *ActivateBlc*
PartialMethDecl	::=	(`before`|`after`)? *MethodDecl*
ActivateBlc	::=	(`activate`|`deactivate`) `{` *BlockStmt*$*$ `}`

Layer declarations in EventCJ have basically the same semantics as the layers in ContextJ. A layer declaration contains a set of partial methods. Similar to ContextJ, a partial method runs before/after the execution of the original method when it has a `before`/`after` modifier, respectively. If a partial method has no modifier, it (called an *around* partial method) runs instead of the original method. An around partial method can execute another around method or the original method by calling the `proceed` pseudo method.

The layer declarations in EventCJ differ from those in ContextJ in the following ways. First, in EventCJ, a layer can declare the `activate`/`deactivate` block that specifies the statements that should be executed whenever the layer is activated/deactivated, respectively. For example, the initialization and termination of the GPS receiver in the pedestrian navigation system can be specified by declaring `activate` and `deactivate` blocks in the `GPSNavi` layer as follows:

```
class Navigation ... {
  MyLOcationOverlay overlay;
  layer GPSNavi {
    activate {
      overlay.onProviderEnabled(...);
      overlay.enableMyLocation();
    }
    deactivate {
      overlay.onProviderDisabled(...);
      overlay.disableMyLocation();
    }
    ... /* other declarations */
  }
}
```

Second, in EventCJ, a layer is not first class. Instead of supporting first-class layers, EventCJ offers an alternative mechanism to control layer activation in a more declarative manner, which is explained in the next section.

3.2 Specifying Layer Transitions

A novel feature of EventCJ is that we can declaratively specify *when* layer transition occurs, *which layers* are activated/deactivated at that time, and *which instances* are affected by the layer transition. This specification is done using event declarations and layer transition rules that are defined separately from the class and layer declarations.

3.2.1 Event Declarations

In EventCJ, an event triggers layer transitions. Syntax of event declaration is as follows:

EventDecl	::=	`declare event` *ID* `(` *ParamList* `)` `:` *Selector* (`:` *SendTo*)?
Selector	::=	*SelectorItem* (`||` *SelectorItem*)$*$
SelectorItem	::=	(`before`|`after`) *PointCut*
SendTo	::=	`sendTo` `(` *Expr* (`,` *Expr*)$*$ `)`

A `declare event` statement consists of three parts. Followed by an event name with a formal parameter list, it declares an *event selector* and a `sendTo` clause. An event selector specifies when this event is triggered, and a `sendTo` clause specifies which instances are affected by this event.

An event selector consists of one or more selector items separated by the logical OR operator `||`. A selector item consists of a modifier and a pointcut. The syntax of a pointcut is simply a subset of that of AspectJ. We can currently use the `call`, `execution`, `set`, and `get` pointcut designators to specify the kinds of actions that cause the event, `target`, `this`, and `args` designators to bind values to parameters declared in the formal parameter list, and the `if` designator to make the pointcut conditional. Those pointcuts are written with either `before` and `after` modifiers in order to specify when the event shall be fired before or after the execution of an action matching the pointcuts. The semantics is also similar to the pointcuts in AspectJ, except that, in EventCJ, each selector specifies the *point in time* [25] during program execution, in the same way as symbols in tracematch [1].

The `sendTo` clause takes a comma-separated list of expressions, each of which is evaluated to an object value. Those expressions can use the formal parameters of the event. Thus, in the `sendTo` clause, we can specify instances accessible from the execution context of the join-point when the event is triggered. When an event declaration has no `sendTo` clause, the event becomes global. This means that every instance, including the one that will be created after the event, is affected by the event. When a layer is controlled by global events, we call it a global layer.

For example, the following statement declares `SwitchDevice`, which will be fired just after each call to the `onStatusChanged` method in the `Navigation` class. It binds a receiver object of a call to the `onStatusChanged` method to the formal parameter `navi` by using the `target` designator, and sends the event to that instance bound to `navi`.

```
declare event SwitchDevice(Navigation navi)
  :after call(void Navigation.onStatusChanged(..))
     &&target(navi)
  :sendTo(navi);
```

3.2.2 Layer Transition Rules

A layer transition rule activates and deactivates the layers of the object that receives the specified event. Its syntax is as follows:

Transition	::=	`transition` *ID* : *Rules*
Rules	::=	*Rule* | *Rules* `|` *Rule*
Rule	::=	*Condition Operator NewContext*
Condition	::=	*ID* | `not` *ID* | `*`
Operator	::=	`switchTo` | `activate`
NewContext	::=	*ID* | `.`

After specifying an event name, layer transition is specified using either the `switchTo` or `activate` operators. The left-hand side of both operators is a condition to apply the rule. When the positive (i.e., without `not`) layer is active, or the negative (i.e., with `not`) layer is inactive on the object receiving the specified event, the rule can be applied. The right-hand side of the operators specifies the layers to be activated. The difference between the `switchTo` and `activate` operators is that the former operator deactivates the positive layers specified on the left-hand side, while the latter retains them. A layer transition rule can have more than one operation separated by `|`. In such a case, the left-most applicable operation will be executed.

The following lines are an example:

```
transition GPSEvent:
  WifiNavi switchTo GPSNavi |
  not OnBoard activate GPSNavi;
```

This transition rule for `GPSEvent` means "if `WifiNavi` is active, deactivate it and activate `GPSNavi`; otherwise, if `OnBoard` is not active, activate `GPSNavi`."

As indicated by the above syntax, we can use a wildcard (`*`) on the left-hand side, which indicates any (possibly empty) set of active layers. This feature is useful to make an operation unconditional. For example, the following transition rule means "deactivate all the active layers and activate `OnBoard`."

```
transition Boarding: * switchTo OnBoard;
```

Furthermore, we can use a dot (`.`) on the right-hand side to represent no active layers. This feature is necessary to simply deactivate the specified layer. For example, the following transition rule means "if `OnBoard` is active, deactivate it."

```
transition Arriving: OnBoard switchTo .;
```

Note that we can declare multiple events at the same join points:

```
event Standby: call(* PowerButton.pressed());
event Wakeup: call(* PowerButton.pressed());
```

When multiple events are triggered at the same join point, and/or when an event is sent to multiple objects, there are multiple layer transition rules that can possibly be applied. In this case, all of the applicable rules *at that time* are applied. For example, with the following transition rules:

```
transition Standby: not LightOn activate LightOn;
transition Wakeup: LightOn switchTo .;
```

when the first rule is applied, then the precondition of the second rule eventually holds after the transition. However, because the rules are applied *just after the events are triggered*, in this case only the first rule is executed. The compiler should report an error when there are conflicted rules such as

```
transition Standby: * activate LightOn;
transition Wakeup: LightOn switchTo .;
```

In EventCJ, an object can have more than one active layer at the same time. For example, with the following layer transition rules, both `WifiNavi` and `Starbucks` are active after occurrences of `WifiEvent` and `EnterSB` (but not `Reset`).

```
transition WifiEvent: * activate WifiNavi;
transition EnterSB: * activate Starbucks;
transition Reset: * switchTo .;
```

The activation of a layer that is already active has no effect on the system (except for the order of active layers). The activation of a currently active layer (we call it *reactivation* of a layer), however, affects the dispatching order of partial methods. In EventCJ, partial methods in the more recently (re)activated layer have higher priority. When `EnterSB` happens after `WifiEvent`, a `before` method in `Starbucks` runs before a `before` method in `WifiNavi`, and an after method in `WifiNavi` runs before an after method in `Starbucks`. When `WifiEvent` is thereafter raised, the order of partial methods is reversed. This mechanism for controlling multiple active layers holds regardless to say that events are declared as a per-instance basis or globally (i.e., whether or not events are declared with the `sendTo` clause).

3.3 Pedestrian Navigation System in EventCJ

We demonstrate the usage of EventCJ by rewriting the pedestrian navigation system in EventCJ.

Similar to the implementation in ContextJ, the EventCJ implementation represents behavioral variations as layers, namely, `GPSNavi` and `WifiNavi` in Figure 4. These provide context-dependent behaviors for GPS-based navigation and wireless LAN based navigation, respectively. Each layer defines a partial method `run` that updates the display. The behavior of this method is different in different layers. In `GPSNavi`, it scrolls the map to update its current position. In `WifiNavi`, it updates the floor plan. The call to `run` occurs when `overlay` detects a new position. Since `run` is achieved as a set of partial methods, the caller of `run` can show a map or floor plan without concern for which and how positioning devices should be used in the current environment.

The `activate` and `deactivate` blocks are useful for performing the initialization and finalization operations of positioning devices. Since either the GPS or wireless LAN device will be turned on and off regardless of the control flow of the system, we should otherwise write those operations at the beginning of every method accessing those devices.

Figure 5 shows the event declarations and layer transition rules for the navigation system[2]. There are two events that will switch instance-specific layers, namely, `GPSEvent` and `WifiEvent`, and two events that will switch global layers, namely, `Boarding` and `Arriving`. The former two declarations extract a `Navigation` instance by using the `target` pointcut and specify it in the `sendTo` clause.

The former two events are generated when a `Navigation` (that implements the `LocationListener` interface declared

[2]Currently, they are defined in a module called *direction*.

```
class Navigation extends MapActivity
    implements Runnable, LocationListener {
  MapView mapView;
  MyLocationOverlay overlay;
  WifiManager wifiManager;
  BuildingGuide buildingGuide;

  void onStatusChanged(...) {...}
  void run() {}
  void onCreate(Bundle status) {
    ... overlay.runOnFirstFix(this); ... }

  layer GPSNavi {
    activate {
      overlay.onProviderEnabled("gps"); }
    deactivate {
      overlay.onProviderDisabled("gps"); }
    after void run() {
      Location loc = overlay.getMyLocation();
      mapView.getController().animateTo(loc);
    }
  }
  layer WifiNavi {
    activate{
      overlay.onProviderEnabled("network");
      wifiManager.setWifiEnabled(true); }
    deactivate {
      overlay.onProviderDisabled("network");
      wifiManager.setWifiEnabled(false); }
    after void run() {
      Location loc = overlay.getMyLocation();
      buildingGuide.updateFloorPlan(loc);
    }
  }
  ...
}
```

Figure 4: Layer declarations in EventCJ.

```
direction SwitchPositioningDevice {
  declare event GPSEvent(Navigation n, int s)
    :after call(void Navigation.onStatusChanged(s))
      &&target(n)&&args(s)
      &&if(s==LocationProvider.AVAILABLE)
    :sendTo(n);
  declare event WifiEvent(Navigation n, int s)
    :after call(void Navigation.onStatusChanged(s))
      &&target(n)&&args(s)
      &&if(s==LocationProvider.OUT_OF_SERVICE)
    :sendTo(n);
  declare event Boarding()
    :after call(void *.cabinModeEntered());
  declare event Arriving()
    :after call(void *.cabinModeExit());

  transition GPSEvent:
    WifiNavi switchTo GPSNavi |
    not OnBoard activate GPSNavi;
  transition WifiEvent:
    GPSNavi switchTo WifiNavi |
    not OnBoard activate WifiNavi;
  transition Boarding:
    * switchTo OnBoard;
  transition Arriving:
    OnBoard switchTo .;
}
```

Figure 5: Event declarations and layer transition rules in EventCJ.

in the Android SDK) instance is notified that the GPS status is changed and the `onStatusChanged` method is called. The `if` pointcut makes these events conditional. These conditions ensure that `GPSEvent` is fired when the GPS system becomes available, and that `WifiEvent` is fired when the GPS system goes out of service. There are also two global events `Boarding` and `Arriving` that are triggered when the user is boarding and disembarking the plane, respectively. Each of them is declared as a global event so that it is broadcasted to the entire system.

Following to the event definitions, a layer transition rule is defined for each event name. The first two rules for `GPSEvent` and `WifiEvent` basically specify that the `WifiNavi` and `GPSNavi` layers alternate with each other unless the device is in the `OnBoard` state. The remaining two rules specify that the device should go to the `OnBoard` state after `Boarding` until `Arriving`, which effectively prevents the device from activating the positioning systems while the device is on an airplane. Note that the last rule merely deactivates the `OnBoard` layer upon `Arriving`, which means that the positioning systems are not automatically reactivated in this specification[3].

As a result of the layer transition, a set of active layers of each instance is changed, and thus, the behavior of it is changed. Therefore, the result of the `run` method changes with respect to the user's surrounding environment; when it is used out of doors, the `GPSNavi` layer is active and the `run` method displays a map of the area obtained by the GPS; when it is used inside a building, it displays a floor map of the building, because the `WifiNavi` layer is active; when the user is boarding the plane, both the GPS receiver and wireless LAN device are automatically switched off because both `WifiNavi` and `GPSNavi` become inactive.

3.4 Discussion

EventCJ textually and logically separates layer activation from the base program. By textual separation, we mean that EventCJ enables us to specify when and what layer activation should take place in a separated module, which had to be achieved by inserting a `with` block into the base program in the existing COP languages. By logical separation, we mean that EventCJ can activate and deactivate layers in a per-instance manner, regardless of the control structure of the base program. This is in contrast with most existing COP languages, which can only activate or deactivate layers within dynamic scope of block statements. Logical separation is suitable to a program that has multiple and/or a

[3]In the current EventCJ, we have no means of saving and restoring active layers. Such a feature would be useful, but it makes verification very difficult.

fragmented thread of control. Multi-threaded programs and programs written on top of a framework are typical examples.

One of the drawbacks of textual and logical separation is its thin connection between layer activation and control structures in the base program. For example, even if the programmer wants to activate a layer during the execution of a method, it is not obvious if the transition rules in EventCJ realize such activation. In the next section, we will see a potential approach to alleviating this problem by using a model-checking technique.

4. VERIFICATION OF LAYER TRANSITIONS

Since layer transition rules form state transition machines, it is easy to translate the rules into model specification languages such as Promela [21]. Translated models can be checked by a model checker so that the models satisfy some expected properties written as a temporal logic formula by the programmer.

Here, we will check the following properties in the layer transition rules for the pedestrian navigation system. First, the `WifiNavi` layer, the `GPSNavi` layer, and the `OnBoard` layer never become active in the same time. Second, after the `Boarding` event, both `GPSNavi` and `WifiNavi` never become active until the `Arriving` event.

We check whether the layer transition rules satisfy such properties when the rules are applied to the base program. For this purpose, we require a model description of the layer transition rules and the base program that can be checked by a model checker and formally-written properties. EventCJ automates translation from layer transition rules into a process in Promela. The model description of the base program is also given by Promela's processes. We assume that this description is given by the programmer as a specification of the system, which is used for verifying not only the layer transition rules, but also for verifying the base program itself. The properties that the layer transition rules should satisfy are formally given by linear temporal logic (LTL) formulae.

The top half of Figure 6 shows the model in Promela translated from the layer transition rules shown in Figure 5. The layer transition rules are translated into one process called `Navigation` that repeatedly listens to a channel called `channel`. Each transition rule corresponds to a message receiving statement with a pattern specifying the event name (e.g., `channel ? WifiEvent` at line 10 in Figure 6). The transition rules connected by the "`|`" operator are translated to an `if` statement (e.g., lines 11–19 in Figure 6). There, the left-hand side of each specification is translated to a guard that checks the activation of layers (e.g., `gps==Active`), and the right-hand side to an atomic action that changes the activation of layers (e.g., `atomic { gps=Inactive; wifi=Active }`). The labels `S0` through `S3` are inserted to specify the states just after the reception of events.

The bottom half of Figure 6 shows a process `Env` that represents the base program and its running environment provided by the programmer. It is an assumption that guarantees that the events `WifiEvent` and `GPSEvent` are triggered at any time, while the `Arriving` event never occurs unless there is one preceding `Boarding` event, and once the `Boarding` event is triggered, it is not triggered until the succeeding `Arriving` event is triggered.

```
 1 mtype = { WifiEvent, GPSEvent, Boarding,
 2           Arriving, Active, Inactive };
 3 chan channel = [0] of { mtype };
 4 mtype wifi = Inactive;
 5 mtype gps = Inactive;
 6 mtype onBoard = Inactive;

 8 active proctype Navigation() {
 9   do
10   :: channel ? WifiEvent ->
11 S0: if
12     :: (gps==Active) ->
13       atomic { gps=Inactive; wifi=Active }
14     :: (onBoard==Inactive) ->
15       atomic { wifi=Active }
16     fi
17   :: channel ? GPSEvent ->
18 S1: if
19     :: (wifi==Active) ->
20       atomic { wifi=Inactive; gps=Active }
21     :: (onBoard==Inactive) ->
22       atomic { gps=Active }
23     fi
24   :: channel ? Boarding ->
25 S2: atomic { wifi=Inactive;
26       gps=Inactive; onBoard=Active }
27   :: channel ? Arriving ->
28 S3: atomic { onBoard=Inactive }
29   od
30 }

32 active proctype Env() {
33   do
34   :: channel ! WifiEvent
35   :: channel ! GPSEvent
36   :: channel ! Boarding ->
37     do
38     :: channel ! WifiEvent
39     :: channel ! GPSEvent
40     :: channel ! Arriving -> break
41     od
42   od
43 }
```

Figure 6: Promela code translated from the layer transition rules.

After building a model, we write the verification properties as LTL formulae in SPIN, such as

```
[]!(inRFNavi && inGPSNavi && inOnBoard)
[]!(afterBoarding V (
  (inGPSNavi || inRFNavi) U afterArriving))
```

where `[]`, `V`, and `U` denote the temporal operators "globally," "release," and "until," respectively. The predicates `inRFNavi`, `inGPSNavi`, `inOnBoard`, `afterBoarding`, and `afterArriving` are defined as below[4]:

[4]Our verification checks properties that all instances should satisfy. Thus, the formula `[]!(inRFNavi && inGPSNavi)`, for example, is interpreted as "for all instances i, `RFNavi` and `GPSNavi` never become active at the same time in i."

```
#define inRFNavi  rf==Active
#define inGPSNavi gps==Active
#define inOnBoard onBoard==Active
#define afterBoarding DeviceController@S2
#define afterArriving DeviceController@S3
```

By running the SPIN model checker, we can confirm that the translated model satisfies the above properties. Even though the example is simple, verifying the above properties is non-trivial. For example, the latter property does not hold if the specification of the base program is faultily given, such as "`WifiEvent`, `GPSEvent`, `Arriving`, and `Boarding` are triggered at any time," indicating that this model checking approach can also detect a fault in the specification.

5. IMPLEMENTATION

We implemented a prototype compiler for EventCJ. It translates an EventCJ source program into an AspectJ program that will be executed with the EventCJ runtime library for managing active layers[5]. The compiler is built on top of the Spoofax/IMP language workbench [24] with the Java-front [9] and AspectJ-front [8] packages. The size of the compiler amounts to 1,078 lines of code.

Directions, event declarations, and layer transition rules are translated into aspects, pointcuts, and pieces of advice, respectively. Layers are translated into inner classes. Globally activated layers are managed using a class variable, while layers activated on an object are managed by itself, i.e., each EventCJ class is translated into a Java class with an array that contains active layers.

We chose, as the translation target language, AspectJ rather than COP languages like ContextJ and JCop. COP languages would be alternative target languages, because implementing EventCJ requires both AOP features to generate events based on the pointcut-like event selectors and COP features to dispatch layer methods. However, it turned out that translating into existing COP languages is not trivial because they do not support per-instance layer activation.

Below, we first explain the translation from EventCJ definitions to AspectJ aspects and Java classes, and then present the runtime overheads of method dispatching in our implementation.

5.1 Translation to AspectJ

The compilation rules are defined mostly modularly for each EventCJ construct. Directions, event declarations, layer transition rules, layers, and partial methods are translated into aspects, named pointcuts, advice declarations, inner classes, and their instance methods, respectively.

In this section, we explain the rules for event declarations, layer transition rules, layers, and partial methods.

5.1.1 Event Declarations

Each event declaration is translated into two AspectJ's named pointcuts. Given an event declaration:

`declare event` $E(\overline{T}\ \overline{x})$`:` s `:` `sendTo(`$\overline{e}$`);`

the compiler separates the event selector s into two groups of selector items, namely, those prefixed by the `before` modifier s_b and `after` modifier s_a. The compiler then generates a named pointcut $E_b(\overline{T}\ \overline{x})$ (respectively, $E_a(\overline{T}\ \overline{x})$) whose body is s_b (respectively, s_a).

[5]It is available from our project page at `http://www.graco.c.u-tokyo.ac.jp/ppp/projects/event-based-cop`.

For example, the following event declaration (which is already shown in Figure 5):

```
declare event GPSEvent(Navigation n, int s)
  :after call(void Navigation.onStatusChanged(s))
   &&target(n)&&args(s)
   &&if(s==LocationProvider.AVAILABLE)
  :sendTo(n);
```

is translated into the following pointcut:

```
pointcut GPSEvent_A(Navigation n, int s):
  call(void Navigation.onStatusChanged(s))
  && target(n) && args(s)
  && if(s == LocationProvider.AVAILABLE);
```

Note that the compiler generates merely one named pointcut because the event declaration merely contains an `after call` selector, but no `before` selector.

The `sendTo` clause is simply ignored, because it does not specify any action, but rather specifies on what objects layers are changed. It is dealt with when the compiler generates an advice body that implements the layer transition rules, as is explained later.

5.1.2 Layer Transition Rules

Each layer transition rule is translated into one or two pieces of advice. Given a layer transition rule:

`transition` E`:`t`;`

the compiler generates before and after advice declarations with the named pointcuts $E_b(\overline{T}\ \overline{x})$ and $E_a(\overline{T}\ \overline{x})$, respectively, which are generated from the event declaration of E. Below, we use E as the name of the event and the event declaration that defines it. The body of advice implements the layer transition t. If the event declaration E contains `sendTo(`$\overline{e}$`)`, it changes the set of active layers on each object obtained by evaluating $\overline{e}$. Otherwise, it changes the set of active layers on every live object.

Consider a concrete example. As we have already shown in Figure 5, we have a layer transition rule in response to a `GPSEvent` event:

```
transition GPSEvent:
  WifiNavi switchTo GPSNavi |
  not OnBoard activate GPSNavi;
```

The compiler translates it into a piece of after advice:

```
1 after(Navigation n,int s): GPSEvent_A(n, s){
2   LayerManager lm=n.lm;
3   if(lm.isActive(WifiNavi.ID))
4     lm.deactivate(WifiNavi.ID).activate(GPSNavi.ID);
5   else if(!lm.isActive(OnBoard.ID))
6     lm.activate(GPSNavi.ID);
7 }
```

The named pointcut `GPSEvent_A(n,s)` is shown above. Line 2 gets the layer manager of the `Navigation` object `n` by accessing its instance field `lm`, which is added to the `Navigation` class by the compiler. The receiver object `n` at line 2 comes from the expression specified in the `sendTo` clause of the `GPSEvent` definition. Lines 3–4 and 5–6 implement the rule `WifiNavi switchTo GPSNavi` and `not OnBoard activate GPSNavi`, respectively.

When the specified event is global, the global layer manager `Global` is used to get the layer managers of all of the live objects. For example, the layer transition rule for the event `Boarding` in Figure 5:

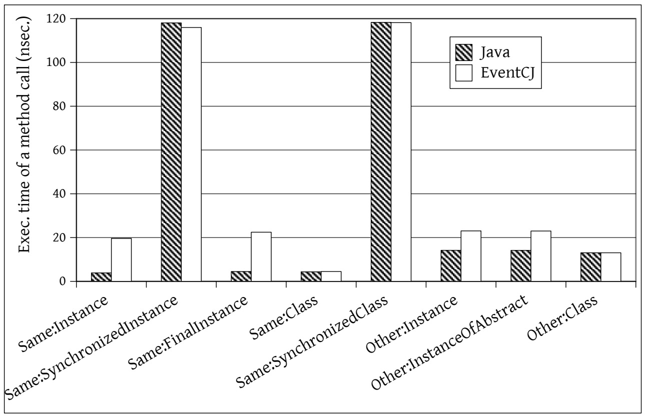

Figure 7: Execution times of a method call in EventCJ and Java (shorter is better).

```
declare event Boarding()
  :after call(void *.cabinModeEntered());
transition Boarding:
    * switchTo OnBoard;
```

is translated into the following after advice:

```
pointcut Boarding_A():
  call(void *.cabinModeEntered());
after(): Boarding_A(){
  for(WeakReference<LayerManager> wlm:
      Global.layerManagers){
    LayerManager lm=wlm.get();
    if(lm!=null)
      lm.deactivateAll().activate(OnBoard.ID);
  }
}
```

Deactivating all active layers and activating the layer `OnBoard` are both performed by the layer manager of each object.

5.1.3 Classes with Layers

We employ basically the same implementation strategy as ContextJ for layers and partial methods, i.e., layers and partial methods are translated into Java classes and methods. The differences are that each translated class has an instance of `LayerManager` to realize the per-instance layer management, and each layer is translated into an inner class.

5.2 Preliminary Performance Measurement

This section evaluates performance of method dispatching in EventCJ by comparing the times for calling methods in EventCJ with and without active layers against the ones for calling plain Java methods. We used a modified version of JGFMethodBench in the Java Grande Forum Benchmark Suite [10] for the measurement. We extended each target method in the program with empty before, after and proceeding around partial methods. All executions are on an Oracle Java HotSpot client VM 1.6.0_22 on four 3.06 GHz 32-bit CPUs running Linux kernel version 2.4.9. In each run, the time is measured after running a measurement method once so as to let the JVM perform optimizations.

Figure 7 summarizes method dispatching performance in Java and EventCJ *without active layers*. The benchmark

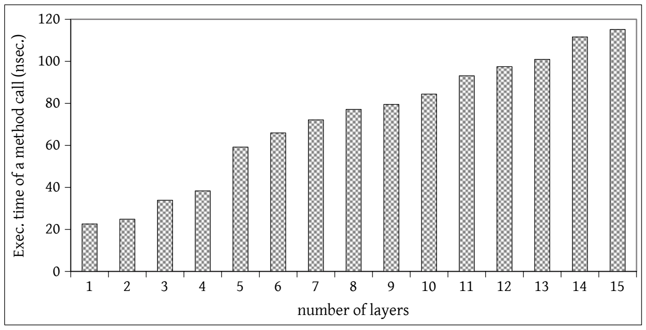

Figure 8: Execution times of a method call in EventCJ when increasing the number of active layers.

program measures execution times of eight kinds method calls. “Same” and “other” mean that the caller and callee methods belong to the same and other instance/class, respectively. “Instance” and “class” mean that the method is an instance and class method, respectively. “Synchronized” and “ofAbstract” mean the method is either synchronized or abstract, respectively. In the EventCJ version, we defined a layer with a partial method for the “instance” methods, which is inactive during the measurements. The “class” methods in EventCJ has no partial methods.

As the figure shows, calls to the “instance” methods in EventCJ are approximately 5 times slower than a call to a plain Java instance method, which can be considered as the overheads of checking active layers. On the other hand, the methods without partial method definitions (the “class” methods) have no overheads relative to Java. Even though the differences in execution environment prevent direct comparison to other COP languages, these overheads factors relative plain Java are similar to the ones in ContextJ [3].

We also measured method dispatching performance in EventCJ *with 1 to 15 active layers*. In this measurement, we define 15 layers, each of which has an empty partial method for the “same:instance” method. Those layers are activated before starting benchmark iterations.

Figure 8 shows the result. As we can see, each additional active layer adds roughly constant time to a call. Again, the result exhibits a similar trend to the one of ContextJ [3].

6. CASE STUDY

In order to assess usefulness of language mechanisms in EventCJ, we carried out a case study that implement two practical application programs in EventCJ. The first program is the Pedestrian Navigation System that we already discussed in the previous sections. The second one is CJEdit, a programming editor originally written in ContextJ [4].

We examine those application programs to assess (1) what layer (de-)activating operations can only be realized by using EventCJ’s mechanisms, and (2) at which program point we need switch layers. Before answering those questions, we first overview CJEdit and its implementations in ContextJ and EventCJ.

6.1 CJEdit Program Editor

CJEdit [4] is a program editor built on top of the Qt/QtJambi framework. It supports two kinds of text fragments, namely, code fragments that are displayed with syntax highlighting and comments that are displayed in the rich text format.

Depending on whether the cursor is on a code fragment or on a comment, CJEdit displays different widgets and renders a program text in different ways as summarized in Table 1. The widgets include an outline view of the program structure, and buttons for changing text properties. Code fragments and comments are rendered normally, with syntax highlighting, and in the grey color.

CJEdit implementations in ContextJ and EventCJ basically declare the same sets of layers for realizing those context-dependent behaviors, as indicated in the typewriter fonts in Table 1. We need to declare two pairs of layers. The `CodeEditing` and `TextEditing` layers reflect whether the cursor is on a code fragment or a comment. The `ActiveHighlighting` and `InactiveHighlighting` reflect a kind of text fragments being rendered.

6.2 Usefulness of EventCJ Constructs

6.2.1 Layer Activation in CJEdit

While the ContextJ and EventCJ implementations basically share the same layer structures, they realize layer activation in different ways.

Switching between the `TextEditing` and `CodeEditing` layers caused by cursor movement to a different kind of text fragment. Since the GUI framework notifies the application program of the cursor movement by invoking a call-back method, it is not possible to activate layers by using the `with` block (without first class layers). The ContextJ implementation therefore uses the workaround, which keeps active layers in a global variable, and wrap all relevant method calls with the `with` block. The EventCJ implementation, on the other hand, simply triggers active layers when the call-back method is invoked.

Selection between the `ActiveHighlighting` and `InactiveHighlighting` layers depends on the kind of text fragment to be rendered and the cursor position. Since it depends on a runtime value (i.e., the kind of text fragment), both of the ContextJ and EventCJ implementations essentially use the same strategy for the selection. In the EventCJ implementation, however, the layer activation code is separated from the base program, while the ContextJ implementation has to insert `with` blocks into the base code.

6.2.2 Use of EventCJ constructs

Finally, we summarize the number of EventCJ constructs used in the two application programs in order to illustrate complexity of context-dependent behaviors. The top-half of Table 2 shows the number of EventCJ constructs used in the pedestrian navigation system (PNS) and CJEdit.

Table 1: Context-dependent behaviors and corresponding layers in CJEdit.

	cursor position	
	code fragment	comment
widgets	outline view	text property
code fragment	syntax highlighting (`ActiveHighlighting`)	plain
comment	grey color (`InactiveHighlighting`)	plain (RTF)
active layer	`CodeEditing`	`TextEditing`

Table 2: Numbers of EventCJ constructs used in context-dependent application programs.

counted constructs	PNS	CJEdit
# layers	3	4
# partial methods	3	10
# transition rules	4	5
# event declarations	4	5
# layer updates	3	0
# `with` blocks	1	3

The bottom-half of the table shows effects of workarounds that would be needed to implement those applications in other COP languages. Each workaround requires to manually manage or compute "active" layers and to activate them whenever context-dependent behavior is to be executed. We therefore assumed (potential) implementations of those application programs in ContextJ, and counted the number of operations that updates "active" layers in a global variable (which shall be zero if the layers can be computed from other global state) as *number of layer updates*, and counted *the number of `with` blocks* inserted in order to actually activate those layers.

Although the number and scale of the application programs are not large, we can observe that EventCJ constructs are useful to separately manage layer activation, which otherwise require workarounds in existing COP languages. We hope that future study will demonstrate that EventCJ gives positive impact on software maintenance processes.

7. RELATED WORK

7.1 Context-Oriented Programming

Most of the existing COP languages are based on a dynamically scoped layer activation mechanism. Since we already discussed ContextJ in detail in Section 2, here we compare other COP languages with EventCJ.

JCop [5] is a successor to ContextJ that can declaratively specify layer activation by using an AspectJ-like pointcut syntax. Although JCop is developed independently of EventCJ, both share the same motivation to separate layer activation descriptions from the base programs. In fact, JCop enables one to describe event specific layer activation from the base program. However, JCop's layer activation is still dynamically-scoped; that is, a layer activated upon a method call can only be deactivated at the end of the call. Therefore, controlling layer activation beyond the dynamic scope of method calls requires a reflection-based workaround, which would make it very difficult to validate safety properties.

Significant implementation efforts on COP languages have been devoted for dynamically typed languages such as Lisp [12], Smalltalk [19], Python [31], and a prototype-based language model named AmOS, whose detailed comparison can be found in other literature [2]. All these COP languages support synchronized layer activation (i.e., the semantics provided by the `with`-block), while EventCJ supports layer activation triggered by asynchronous events.

Asynchronous layer activation is also supported by ContextErlang [18, 17], which is a context-oriented extension to Erlang that supports asynchronous context activation. In ContextErlang, context activation is modeled as a message sent from a supervisor process called context manager; thus, asynchronous context activation is naturally represented. As in EventCJ, the message can be broadcasted, or sent to the processes run on a specified node. Unlike EventCJ, ContextErlang activates and deactivates contexts by executing imperative operations tangled in the base program.

There are extensions to COP languages that can coordinate layer activation based on their dependency and exclusiveness among layers [11, 13]. Though those extensions aim at ensuring similar kind of safety properties, the approaches are totally different. First, those extensions are based on dynamically scoped layer activation mechanisms (i.e., the `with`-block). Second, those extensions provide runtime coordination mechanisms, which can merely stop a program at runtime when conflicting set of layers are to be activated. EventCJ does not provide an explicit means of describing dependency and exclusiveness. Instead, we propose an approach to verify safety properties before executing a program by using a model checker.

From the view point of layer activation per-instance, EpsilonJ [29, 30] provides a similar mechanism, which is called object adaptation, that changes object's behaviors dynamically. Since EpsilonJ only provides the imperative `bind` operation for object adaptation, it is very difficult to perform verification as presented in this paper. NextEJ [23] is a variant of EpsilonJ that supports scoped object adaptation by using the `with`-block. Unlike ContextJ, the `with`-block in NextEJ can specify the instances that are affected by the object adaptation within a specific control flow. Thus, unlike EventCJ, NextEJ cannot control object adaptation asynchronously triggered by events.

7.2 Aspect-Oriented Programming

EventCJ is related to AOP languages in two ways. First, EventCJ's event selectors are based on the pointcut language and its join-point model in AOP (AspectJ, in particular). Second, AOP languages that can dynamically deploy aspects can implement many features that EventCJ provides.

We already discussed AspectJ in Section 2. We now provide two more detailed differences between AspectJ's pointcuts and EventCJ's event selectors. First, event selectors in EventCJ are based on the *point-in-time* join-point model [25], while pointcuts in AspectJ are based on the *region-in-time* model. Second, an AspectJ pointcut specifies when a piece of advice *run*. This means that the advice in AspectJ directly affects the behavior of the join point that matches the pointcut. On the other hand, an event in EventCJ indirectly affects behavior by changing the layer activation of certain objects.

Trace-based extensions to AOP [1, 32] have interesting similarities with EventCJ. First, those extensions also employ the point-in-time join point model to recognize the trace of join points. Second, those trace-based extensions are based on regular expressions, which can recognize the same class of languages that finite state machines can do in EventCJ. However, those trace-based extensions are merely interested in accepted states because they describe a condition of advice execution, while each state (i.e., active layers) is meaningful in EventCJ.

CaesarJ [6] supports dynamic aspect control to deploy aspects globally, locally, and thread-locally as well as statically. Using these constructs, we can encode context-specific behaviors that are dynamically deployed and undeployed so that the behavior of the system can change with respect to the surrounding context. However, operations to dynamically control aspects are imperative in CaesarJ. On the other hand, EventCJ focuses on the declarative specification on when each layer transition occurs, which instance is affected by that layer transition, and which layer becomes active and inactive at the time of layer transition.

CSLogicAJ [28] is a language that supports adaptation of service behavior by using context-sensitive service aspects. It is based on the aspect-oriented language LogicAJ [27]. Similar to EventCJ, context change in CSLogicAJ is modeled in terms of join points. The current implementation of CSLogicAJ is limited to systems on the OSGi framework, and can merely adapt services available in a local OSGi registry.

7.3 Event-based Programming

The notion of events in EventCJ originates from event-based programming, which advocates programs that mainly react to asynchronous events.

Polyphonic C♯ [7], which is now a part of Cω, is an extension to C♯ that supports asynchronous methods. EventJava [16] is an extension of Java that integrates events with methods. In both languages, events are triggered by method calls imperatively. In EventCJ, on the other hand, the time when an event is triggered is declaratively given by selecting join points. Like EventJava, EventCJ supports the broadcasting of events as well as unicasting, but EventCJ also supports the multicasting of events by specifying multiple instances in the `sendTo` clause.

ECaesarJ [26] is an extension to CaesarJ with explicit events and their handlers. Events in ECaesarJ are similar to those of EventCJ in that they declaratively define when the event is triggered. ECaesarJ can simulate event-driven layer transitions by letting event handlers imperatively deploy and undeploy aspects that define context-specific behaviors. In this sense, ECaesarJ is similar to the AspectJ-based approach explained in Section 2.

8. CONCLUSION

The event-based layer transition mechanism in EventCJ makes it possible to declaratively specify layer activation apart from the base program. By declaring events in a pointcut-like language, we can flexibly specify when layer transitions should occur in a separate manner. Per-instance based layer activation can affect behaviors of methods executed by different threads in a finer-grain. These features contribute better modularity to context-aware programs whose context changes occur upon internal or external events. Furthermore, we demonstrated an approach to verify basic properties of layer transition rules, which first translates those rules to Promela's process definitions, describes the base program's model in Promela, writes verification properties in linear temporal logic, and finally executes the SPIN model checker. We implemented a prototype compiler of EventCJ, which demonstrated reasonable performance when compared with other Java-based context-oriented programming languages.

Acknowledgments.

We thank all the members of the Kumiki project for their fruitful discussion on the design of EventCJ. Tsutomu Kumazawa, Lennart Katz, and Malte Appeltauer and Robert Hirschfeld gave us helpful information on model checkers, the Spoofax language workbench, and ContextJ and JCop, respectively. We also appreciate the valuable comments and suggestions from the anonymous reviewers. This work is supported in part by a Grant-in-Aid for Scientific Research No.22240002 and a Grant-in-Aid for Young Scientists (B) No.20700022 from MEXT of Japan.

9. REFERENCES

[1] Chris Allan, et al. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*, pages 345–364, 2005.

[2] Malte Appeltauer, et al. A comparison of context-oriented programming languages. In *COP '09*, pages 1–6, 2009.

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 2011. to appear.

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

[5] Malte Appeltauer, et al. Event-specific software composition in context-oriented programming. In *SC'10*, pages 50–65, 2010.

[6] Ivia Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *TAOSD-I*, pages 135–173, 2006.

[7] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *TOPLAS*, 26(5):769–804, 2004.

[8] Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for AspectJ. In *OOPSLA '06*, pages 209–228, 2006.

[9] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA'04*, pages 365–383, 2004.

[10] Mark Bull, Lorna Smith, Martin Westhead, David Henty, and Robert Davey. Benchmarking Java Grande applications. In *PA-Java'00*, pages 63–73, 2000.

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

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

[13] Pascal Costanza and Robert Hirschfeld. Reflective layer activation in ContextL. In *SAC'07*, pages 1280–1285, 2007.

[14] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient layer activation for switching context-dependent behavior. In *JMLC'06*, pages 84–103, 2006.

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

[16] Patrick Eugster and K.R. Jayaran. EventJava: An extension of Java for event correlation. In *ECOOP'09*, pages 570–594, 2009.

[17] Carlo Ghezzi, Matteo Praella, and Guido Salvaneschi. Context-oriented programming in highly concurrent systems. In *COP'10*, 2010.

[18] Carlo Ghezzi, Matteo Praella, and Guido Salvaneschi. Programming language support to context-aware adaptation–a case-study with Erlang. In *SEAMS'10*, pages 59–68, 2010.

[19] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *GTTSE'07*, pages 396–407, 2008.

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

[21] Gerard J. Holzmann. The model checker SPIN. *TSE* 23(5):279–295, 1997.

[22] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Designing event-based context transition in context-oriented programming. In *COP'10*, 2010.

[23] Tetsuo Kamina and Tetsuo Tamai. Towards safe and flexible object adaptation. In *COP'09*, 2009.

[24] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *OOPSLA'10*, 2010.

[25] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A fine-grained join point model for more reusable aspects. In *APLAS'06*, pages 131–147, 2006.

[26] Angel Nú nez, Jacques Noyé, and Vaidas Gasiūnas. Declarative definition of contexts with polymorphic events. In *COP'09*, 2009.

[27] Tobias Rho, Günter Kniesel, and Malte Appeltauer. Fine-grained generic aspects. In *FOAL'06*, 2006.

[28] Tobias Rho, Mark Schmatz, and Armin B. Cremers. Towards context-sensitive service aspects. In *OT4AML*, 2006.

[29] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *ICSE'05*, pages 166–175, 2005.

[30] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. Objects as actors assuming roles in the environment. In *Software Engineering for Multi-Agent Systems V*, pages 185–203, 2007.

[31] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *ICDL'07*, pages 143–156, 2007.

[32] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *FSE'04*, pages 159–169, 2004.