A Unified Context Activation Mechanism

Tetsuo Kamina
University of Tokyo
kamina@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology
aotani@is.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

ABSTRACT

With the increase of research interest in context-oriented programming (COP), several COP languages with different characteristics have been proposed. Although they share common language features to modularize context-dependent variations of behavior, they take quite different ways to realize them. Because of such differences, each language cannot solely cover all use cases of implementing context-dependent behavioral variations. In this paper, we propose a new COP language Javanese that unifies several COP mechanisms into general linguistic constructs. Specifically, it provides *context declarations* to identify context and its specification of the range of execution sequences where this context is active, *activate declarations* to define the relation between contexts and layers, and *context group declarations* that modularize these declarations and specify the set of instances where they are applied. This paper describes design of Javanese and an implementation strategy.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*

General Terms

Languages

Keywords

Context-oriented programming, EventCJ, Layer activation, Composite layers

1. INTRODUCTION

With the increase of research interest in context-oriented programming (COP), several COP languages with different characteristics have been proposed. These languages take quite different ways to represent dynamic changes of behavior with respect to context changes. Some COP languages activate layers under specific control flows by using `with`-blocks [7, 4, 5]. Some other languages activate layers by events, and how layer activation occurs is specified by transition rules [12]. In some COP languages, the effect of switching of contexts (layers) is global [8], while in other languages, such effect is restricted to a specific set of instances [12], or a specific execution thread [7, 4, 5].

There are several tradeoffs between current COP mechanisms. For example, `with`-blocks are convenient to restrict activation of layers within a specific control flow; however, they are not suitable to represent activation of layers beyond control flows. Specifying layer activation *per-instance* is convenient to represent fine-grained control of behavioral changes (e.g., instances of the same class may take different behavior at the same time), while the global layer activation allows us to represent more drastic changes of behavior that affect the whole application.

Despite these differences, we observed that these languages share a set of common goals: with respect to context changes, all COP languages specify the range of execution sequences where some execution entities (such as objects and threads) are adapted to acquire new behavior. Most of them modularize such variations of behavior using layers, and all of them identify, implicitly or explicitly, contexts and their relation to behavior. In other words, each COP language can represent only some specific cases of context-dependent behavioral changes.

In this paper, we propose a new COP language Javanese that unifies several COP mechanisms into general linguistic constructs. Specifically, Javanese realizes a context as a property of the system that is activated by an action and held active until another action that deactivates it occurs. A context in Javanese is defined as a term of temporal logic with a call stack, which can represent most of existing layer activation mechanisms. Furthermore, besides existing layer activation mechanisms that explicitly specify the action that activate the layer, Javanese also supports implicit activation of contexts and layers.

Javanese is a layer-based COP language, which provides the modularization mechanism for context-dependent behavior using layers. Unlike most of existing layer-based COP languages, Javanese provides *activate declarations* that relate a layer with a proposition where ground terms are contexts (true when active). A layer is implicitly activated when that proposition becomes true. In other words, in Javanese, all layers are composite layers [14].

Javanese also provides an abstraction mechanism for representing a set of objects that are affected by layer activation. In Javanese, contexts and activate declarations are grouped into a *context group declaration*, which specifies objects whose behavioral change is controlled by the enclosed activate declarations. An object can subscribe/unsubscribe context groups at runtime, and only the subscribers are controlled by the context group. When objects subscribe (and unsubscribe) that context group is specified by using an AspectJ-like pointcut language. Furthermore, we can also declare a *global context group* that affects all objects in the application.

The rest of this paper is organized as follows. Section 2 reviews existing COP mechanisms, and discusses tradeoffs between them. Section 3 sketches the design of Javanese. Section 4 discusses implementation issues. Section 5 discusses related work. Finally, Section 6 concludes this paper and delivers future directions of research.

2. EXISTING COP MECHANISMS

In this section, we review existing COP mechanisms from the following viewpoints: how they select the range of execution sequences, how they specify targets of activation, and how they identify "contexts" and relate them to behavior[1].

2.1 Selecting Range of Execution Sequences

Specifying blocks. One of the most common ways to activate context-dependent variations of behavior is using `with`-blocks, which activates specified layers only within the dynamic scope of the block [7, 4, 5]. For example, by taking an example of pedestrian navigation system that changes its behavior with respect to situations such as outdoors and indoors, we can activate the layer `Outdoors` (that defines behavioral variations that are executable only when that system is in outdoors) by using the following `with`-block:

```
with (Outdoors) { .. }
```

By explicitly specifying the block-structured scope of activation, `with`-blocks make it easy to reason about some desirable properties (e.g., there are no unintentional collisions between layers).

Specifying definitive activation. Some COP languages provide *definitive activation*, which is realized by imperative operations to activate behavior that definitively affects the rest of execution [8, 9]. For example, in Subjective-C [8], activation and deactivation of contexts are written as follows:

```
[CONTEXT activateContextWithName: @"Outdoors"];
[CONTEXT deactivateContextWithName: @"Indoors"];
```

The first line activate the context `Outdoors`, and the second line deactivate the context `Indoors`.

Specifying declarative layer switching. EventCJ [12] supports event-driven layer switching, which activates/deactivates layers upon generation of events. An event is declaratively defined using an AspectJ-like pointcut language:

```
event GPSEvent(int s)
  :after call(void Nav.onStatusChanged(*))
    && args(s) && if(GPS.AVAILABLE==s);
```

This event declaration specifies that the event `GPSEvent` is generated just after the call of `onStatusChanged` method declared in the class `Nav`, and if the class field `GPS.AVAILABLE` is equal to the argument for `onStatusChanged` method call.

The layer switching upon event is declaratively specified by using the following layer transition rule:

```
transition GPSEvent:
  Indoors ? Indoors -> Outdoors | -> Outdoors;
```

This rule is interpreted as follows: when `GPSEvent` is generated, if the layer `Indoors` is active, then it is deactivated and `Outdoors` is activated; otherwise, no layers are deactivated and `Outdoors` is activated.

Unlike `with`-blocks, this approach enables us to represent layer activation beyond control flows. Furthermore, by providing the model-checking mechanism to verify safety properties of layer transitions, EventCJ compensates for the loss of disciplined layer activation enforced by the block-structured layer activation control.

2.2 Selecting Targets of Activation

Per-thread. One of the most common *targets* to apply context-dependent behavioral variations are current executing threads. The `with`-blocks inherently affect executing threads, because they change the behavior of control flows. Another example of this *per-thread* activation can be found in ContextErlang [16], which is a COP extension to an actor-model programming language Erlang. In ContextErlang, we specify the process (that corresponds to an executing thread) where the context-dependent behavioral variations are applied (in this sense, it is also considered as per-instance activation, which is discussed below).

Per-instance. In COP languages that do not activate layers by using `with`-blocks, the targets to apply context-dependent behavior are not restricted to executing threads. For example, EventCJ activates layers in the *per-instance* manner. In event declarations in EventCJ, we can specify objects to which the specified event is sent:

```
event GPSEvent(Nav n, int s)
  :after call(void Nav.onStatusChanged(*))
    && args(s) && target(n) && if(GPS.AVAILABLE==s)
  :sendTo(n);
```

By using the `sendTo` specification, this event declaration specifies that the `GPSEvent` is sent to the receiver of the `onStatusChanged` method call, which is bound to `n` by using the `target` pointcut. Since layer transition rules are applied only to the receivers of the event, EventCJ controls layer activation per-instance.

Global. Another way to select targets for activation is to activate context-dependent behavior *globally*; i.e., we do not select any targets but the whole application is affected by the activation. This approach is taken by Subjective-C [8] and Ambience [9]. Furthermore, in EventCJ, we can omit the `sendTo` specification in the event declaration, which means that the effect of event is global.

2.3 Layer vs Context

Another viewpoint to characterize COP languages is how they identify contexts and relate them to behavioral variations.

Most COP languages are *layer-based*, where contexts are implicitly identified as layers (modules that enclose context-

[1]This section is a summary of another paper submitted by one of the authors.

dependent behavior). In such languages, we explicitly specify layers to be activated with respect to context changes.

In non-layer-based COP languages such as Subjective-C and Ambience, context-dependent behavior is not modularized using layers. Instead, a context is reified as an object that is implicitly used in the method invocation. Each context-dependent method is defined with a context, which can be combined with other contexts.

Recently, similar composition mechanisms are also proposed in layer-based COP languages [6, 14]. Specifically, *composite layers* mechanism in EventCJ [14] represents a layer as a composition of contexts. In this mechanism, a layer can be specified with a proposition where ground terms are other layers (true when active). Such layer is implicitly activated when that proposition becomes true. Only layers that do not specify such propositions, which are called *atomic layers*, can explicitly be activated by layer transition rules. In this sense, each atomic layer can be considered as a context, and a composite layer defines a relation between contexts and context-dependent behavior.

2.4 Tradeoffs

Each COP language cannot solely cover all use cases of implementing context-dependent behavioral variations. First, `with`-blocks are convenient to restrict activation of layers within a specific control flows. However, they are not suitable to represent activation of layers beyond control flows. For example, in the pedestrian navigation system, layer switching is executed in a callback method when some status changes (such as changes in GPS signal intensity) are detected. On the other hand, in `Outdoors`, we define partial methods `display` that displays a city map (instead of displaying a floor plan, which is the behavior for `Indoors`) and `getPos` that gets current user's location from GPS, which are definitely not called from those callback methods.

Second, the global layer activation allows us to represent drastic changes of behavior that affect the whole application. However, it cannot represent more fine-grained control of behavioral changes. For example, as in the context-aware Twitter client that supports multiple tabs [14], instances of the `Tab` class may take different behavior at the same time.

Finally, layers are convenient to modularize related pieces of code. In general, however, such modules do not directly correspond to more fine-grained contextual information such as running machine's status. Thus, as noted in [14], several pieces of layer activation code are scattered and tangled in the program in layer-based COP languages.

Explicit vs implicit activation of behavior. Besides the aforementioned tradeoffs, we also note that most existing COP mechanisms activate variations of context-dependent behavior *explicitly*. In `with`-blocks, we explicitly specify layers to be activated. In layer transition rules, we directly control activation of layers by specifying the name of layers. Other non-layer-based COP languages also specify the name of contexts in the program. Although composite layers are implicitly activated, all atomic layers still have to be activated explicitly.

However, in some cases *implicit activation* of behavior is more convenient than the explicit one. For example, assume that `StrongGPS` is a context where GPS signal intensity is strong. To determine when `StrongGPS` is active, it is more understandable to declare that "`StrongGPS` becomes active when the value of `GPS.SIGNAL` is higher than a threshold," than to specify the action as follows: "`StrongGPS` becomes active when the callback method that changes the value of `GPS.SIGNAL` and it is over the threshold after the execution of the callback method." The latter is also error prone in particular when there are several such actions.

3. DESIGNING UNIFIED ACTIVATION

Section 2 discusses several COP mechanisms and their tradeoffs. Despite these differences, we observed that such mechanisms share a set of common goals. First, with respect to context changes, all COP languages specify the range of execution sequences where some execution entities such as objects and threads are adapted to acquire new behavior. Second, all COP languages provide a way to specify targets where the specified behavior is adopted. Finally, all COP languages identify, implicitly or explicitly, contexts and their relation to behavior.

In this paper, we propose a new COP language Javanese that unifies several COP mechanisms into general linguistic constructs. Specifically, it provides the following constructs: *context declarations* to identify context and its specification of the range of execution sequences where this context is active, *activate declarations* to define the relation between contexts and layers, and *context group declarations* that modularize these declarations and specify the set of instances where they are applied.

Javanese is a layer-based COP language, which provides the modularization mechanism for context-dependent behavior using layers. Currently, the syntax of layer is exactly the same as that in EventCJ [12], which provides layer-in-class style of layer declarations where we can define a set of partial methods and `activate`/`deactivate` blocks. This paper focuses on the mechanisms for layer *activation*, and the style of layer *declaration* is considered out of scope.

3.1 Context Declarations

In Javanese, a context is defined as a property of the system that is activated by an action and held active until another action that deactivates it occurs. In Javanese, a context is declared with a term of temporal logic with call stacks, which consists of *active-until expressions* that specify the active-event and until-event that activate and deactivate context respectively, *if expressions* that specify the condition when that context is active, and *cflow expressions* that specify the control flows where that context is active.

The syntax of context declaration is as follows:

`context` *ContextName Term* ;

It starts with the keyword `context` followed by the name of context. The detailed syntax for terms to specify when that context is active is discussed in the subsequent sections.

Active-until expressions. An active-until expression specifies events that activate and deactivate the context by using AspectJ-like pointcut language. For example, the context `GPSon`, which is activated just after the call of `onStatusChanged` method and if the value of `GPS.AVAILABLE` is equal to the argument for the method call, and deactivated just after the call of the same method and if the value of `GPS.AVAILABLE` is not equal to the argument, is defined as follows:

```
context GPSon
  active(int s) :after call(
    void Nav.onStatusChanged(*)) && args(s)
    && if(GPS.AVAILABLE==s)
  until(int s) :after call(
    void Nav.onStatusChanged(*)) && args(s)
    && if(GPS.AVAILABLE!=s);
```

An active-until expression specifies an *active-event*, which activates the context, and an *until-event*, which deactivates the context. The specification of active-event starts with the keyword `active` followed by the list of local variable declarations that are bound in the pointcut expressions. The pointcut expressions are written with either `:before` or `:after` modifiers in order to specify when the event shall be fired before or after the execution of an action matching the pointcuts. Thus, these events are similar to those in EventCJ [12]. An until-event is also specified in a similar way.

Cflow expressions. Besides event-based representation realized by the active-until expressions, Javanese also supports context activation in the per-control-flow style. This is realized by the following cflow expression:

```
context RouteSearching
  cflow(call(void Nav.calcRoute()));
```

This context declaration specifies that the context `RouteSearching` is active only under the control flow specified by the `cflow` expression, which is the whole execution of the `calcRoute` method declared in the `Nav` class. Note that cflow expressions are not a particular case of active-until expression when the specified call is recursive. In the case of recursion, a cflow expression activates the context during the execution of the first call of the specified method.

Conditional expressions. As discussed in Section 2, implicit activation of contexts is sometimes more convenient than the explicit one. To support implicit activation, Javanese provides the if expressions that specify the condition when the context is active. For example, the abovementioned context `GPSon` can be declared by using the if expression as follows:

```
context GPSon if(GPS.AVAILABLE==true);
```

In the if expressions, we can use any boolean-type Java expressions. We can also use variables declared in the enclosing context group declaration (discussed in Section 3.3).

3.2 Activate Declarations

Unlike most existing layer-based COP languages, in Javanese, the activation of layers cannot be controlled explicitly. Instead, it provides a way to declare when the layer is active in terms of contexts by using *activate declarations*. An activate declaration assigns a proposition where ground terms are names of contexts (true when active) to a layer. The layer is active when this proposition is true. For example, assuming that the layer `Outdoors` is active only when both contexts `GPSon` and `StrongGPS` are active, we specify the activate declaration for `Outdoors` as follows:

```
activate Outdoors when GPSon && StrongGPS;
```

This declaration starts with the keyword `activate`, which is followed by the name of layer whose activation is controlled by this declaration. The condition is specified in the `when` clause. In this condition, we can use the logical operators `||`, `&&`, and `!` to compose contexts.

```
1  contextgroup PNav(GPS gps) pertarget(GPS.new(..)) {
2    subscribers(Nav nav, GPS g) :
3        call(void Nav.setGPS(GPS)) && target(nav)
4        && args(g) && if(g==gps)
5      { subscribe(nav); }

7    context GPSon
8      active(int s) :after call(
9        void Nav.onStatusChanged(*)) && args(s)
10       && if(GPS.AVAILABLE==s)
11     until(ChangeEvent e) :after call(
12       void Nav.onStatusChanged(*)) && args(s)
13       && if(GPS.AVAILABLE!=s);

15   context StrongGPS
16     if(GPS.SIGNAL>=gps.currentValue());

18   activate Outdoors when GPSon && StrongGPS;
19   activate Indoors when !GPSon || !StrongGPS;
20 }
```

Figure 1: Context group declaration for the pedestrian navigation system

3.3 Context Group Declarations

To modularize related context and activate declarations, Javanese provides *context group declarations*. More precisely, a context group declaration provides three functions: to group related declarations into one module, to specify the set of instances where these declarations are applied, and to specify instances that the context group observes.

An example of context group declaration is shown in Figure 1. The line 1 specifies the name of context group and how it is instantiated. The `pertarget` clause specifies that the context group `PNav` is instantiated when the instance of `GPS` is created, and this context group observes the *target* of this instance creation (in this case, the created instance), which is bound to the variable `gps`. Currently, Javanese supports only `pertarget` and `perthis` clauses for context group instantiation. More expressive mechanisms to specify a set of related instances remains as future work, though we assume that the mechanism like association aspects [15] may be applied for this purpose. If no `pertarget` and `perthis` clauses are used in the context group declaration, this context group is *singleton*; i.e., it is created at the beginning of execution of the application, and remains until termination of the application. A singleton context group does not observe any objects.

Lines 2-5 in Figure 1 specify the set of instances where enclosed context and activate declarations are applied. These instances are called *subscribers* of this context group. In Javanese, we declaratively specify subscribers by using AspectJ-like pointcut language. The subscription occurs when the `setGPS` method is called. The line 5 specifies that the target of the `setGPS` method call is subscribed to this context group. In this code block, we can also use the `unsubscribe` statement, which specifies the object to be unsubscribed

```
global contextgroup PNav {
  context GPSon
    active(int s) :after call(
      void Nav.onStatusChanged(*)) && args(s)
      && if(GPS.AVAILABLE==s)
    until(ChangeEvent e) :after call(
      void Nav.onStatusChanged(*)) && args(s)
      && if(GPS.AVAILABLE!=s);

  context StrongGPS
    if(GPS.SIGNAL>=GPS.CURRENT_VALUE);

  activate Outdoors when GPSon && StrongGPS;
  activate Indoors when !GPSon || !StrongGPS;
}
```

Figure 2: Global context group declaration

from this context group.

Note that we can use multiple `subscribe` statements within the code block, and we can use multiple `subscribers` statements within the same context group declaration. Thus, compared with EventCJ where receivers of event can be specified only from the same join-point, we can more flexibly specify the set of instances where context and layer activation is applied.

Lines 7-13 and 15-16 declare contexts `GPSon` and `StrongGPS` by using an active-until expression and an if expression, respectively. Note that we use the variable `gps` declared in line 1 in the if expression. The line 18 specifies that the `Outdoors` layer is active only when both `GPSon` and `StrongGPS` are active, and the line 19 specifies that the `Indoors` layer is active only when neither `GPSon` nor `StrongGPS` are active.

Global context groups. As discussed in Section 2, global layer activation is convenient to represent drastic changes of behavior that affect the whole application. For this purpose, Javanese provides *global context groups* that enclose context and layer declarations applied to every object within the application.

Figure 2 shows the global context group version of the pedestrian navigation system. A global context group does not contain any `subscribers` specifications to specify objects to be subscribed and unsubscribed. Instead, every object is implicitly considered to be subscribed to the global context group. Note that this global context group is singleton, and thus it can observe only global variables such as public class variables. Thus, assuming that the class `GPS` has a class variable `CURRENT_VALUE` that is updated when the current GPS signal intensity is changed, the if expression in the context `StrongGPS` is modified to access this variable.

3.4 Discussion

We discuss how Javanese unifies several COP mechanisms.

First, the active-until expressions and activate declarations simplify layer transition rules in EventCJ, and cover all use cases expressed by them. As discussed in [14], layer transition rules suffer from the scattered and tangling problem, which is addressed by the application of composite layers. Using composite layers, however, it is observed in several case studies that most transitions of atomic layers (i.e., layers directly controlled by events) have just two states, and thus we can simply declare a context instead of a set of transition rules. Some cases where transitions of atomic layers have more than two states can also be represented using active-until expressions, by identifying each state as a context. As in EventCJ, both per-instance and global activations are also supported in Javanese.

Second, context declarations support several kinds of terms. Cflow expressions provide a declarative style of `with`-blocks, which is similar to the syntax of JCop [5]. Although there is a semantical difference between cflow expressions and `with`-blocks in that the latter activate layers per-thread manner, both cover several same use cases to activate layers only under specific control flows. Similarly, a definitive activation is represented by an active-until expression in a global context group. In Javanese, these different mechanisms are integrated in the same linguistic mechanism to declare when the contexts are active using temporal logic.

Finally, we discuss that Javanese is more expressive than existing COP languages. First, context declarations support if expressions. This allows us to represent implicit context activation, which is not supported by most existing COP mechanisms. Furthermore, the `subscribers` specification in Javanese enhances the expressive power of the per-instance activation mechanism in EventCJ, where only instances accessible from the join-point of events can be selected as targets of events. Javanese addresses this problem by separating when these targets are subscribed from when the event that activates a context occurs.

4. NOTES ON IMPLEMENTATION

Basically, a Javanese program is translated into an AspectJ program. In this translation, the active-event and until-event of active-until expressions are translated into corresponding pointcut declarations. Layer activation performed by the activate declarations are translated into advice blocks. This compiler creates a table for each layer that maps combinations of contexts to states of the layer. Cflow expressions are also translated in a similar manner except that, in the advice code, we need to count the depth of call stack to appropriately handle the case when the specified control flow contains recursive calls.

Unlike two other expressions, if expressions are translated into the body of the layered method (i.e., a method that consists of a set of partial methods) where the control of calls of partial methods and the original method is encoded; if the condition in the if expression holds, it delegates the method call to the partial methods, otherwise, it delegates it to the original method.

Implementation of layers is similar to that in EventCJ; a layer is translated into an inner class, and partial methods are translated into method declarations within that inner class. To enable per-instance layer activation, an instance variable that refers to an instance of *layer manager* is introduced to each class that declares layers.

5. RELATED WORK

We firstly introduce relevant COP languages that are not fully discussed in Section 2. Asynchronous layer activation is supported by ContextErlang [16], which is a context-oriented extension to Erlang. In ContextErlang, context activation is modeled as a message sent from a supervisor process called context manager. As in EventCJ, the message can be broadcasted, or sent to the processes run on a specified node. PyContext [18], which is a Python-based framework for context-oriented programming, also supports layer activation that goes beyond specific control flows by providing implicit layer activation.

LEAD/LEAD++ [1, 2] also supports a mechanism similar to implicit layer activation. In this language, a method consists of several implementations, each of which is assigned a condition. Only the implementation where this condition holds is selected to execute. This condition is changed with respect to states of the objects used in that condition, called *metaobjects*. Programmers can change state of the metaobjects through API provided by the language.

Tanter et al. proposed context-aware aspects [17], aspects whose behaviors depend on contexts. This concept is realized as a framework where a context is defined as a pointcut, which is similar to AspectJ's `if` pointcut but also able to restrict the *past* contexts. Contexts are composable, because they are realized as pointcuts.

6. CONCLUSIONS AND FUTURE WORK

This paper summarizes differences and commonalities of existing COP languages, and proposes a new COP language Javanese that unifies several COP mechanisms into general linguistic constructs. It provides context declarations, activate declarations, and context group declarations to support wide variety of existing COP mechanisms into one COP language. Furthermore, it supports implicit activation that has not been supported by most existing COP languages. This solves the tradeoffs between existing COP mechanisms and make Javanese expressive enough to naturally implement almost all COP applications supported by existing COP languages by using one language.

We deliver the directions of future work as follows. First, it is interesting to precisely define the semantics of Javanese. Several COP calculus has been proposed to date, in particular for `with`-blocks [10, 11], event-based layer transition [3], and its extension to composite layers [13]. Javanese unifies several activation mechanisms into one linguistic construct called context declarations. To precisely define the semantics of context and compare it with existing calculus will provide us deeper insight of COP semantics.

Second, in current version of Javanese, any extension mechanisms for context groups are not considered. To make context groups extensible and reusable, it is desirable to facilitate some composition mechanism to context groups, such as inheritance, redefinition (overriding) of contexts and activate declarations, and/or trait-like composition mechanism.

Third and finally, some verification mechanism should be considered in Javanese to compensate for the disciplined layer activation mechanism. The `with`-blocks explicitly specify the scope of activation by using blocks, which makes it easy to reason about some desirable properties. EventCJ supports automaton-based model checking. Similar mechanism should also be considered in Javanese.

7. REFERENCES

[1] Noriki Amano and Takuo Watanabe. LEAD: a linguistic approach to dynamic adaptability for practical applications. In *IFIP TC2 WG2.4 Systems implementation 2000 : languages, methods and tools*, pages 277–290, 1998.

[2] Noriki Amano and Takuo Watanabe. LEAD++: an object-oriented language based on a reflective model for dynamic software adaptation. In *TOOLS 31*, pages 41–50, 1999.

[3] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. In *COP'11*, 2011.

[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, 2011.

[5] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *SC'10*, volume 6144 of *LNCS*, pages 50–65, 2010.

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

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

[8] Sebastián González, Micolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In *SLE'11*, volume 6563 of *LNCS*, pages 246–265, 2011.

[9] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object systems. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

[10] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *FOAL'11*, pages 19–23, 2011.

[11] Atsushi Igarashi, Robert Hirschfeld, and Hidehiko Masuhara. A type system for dynamic layer composition. In *FOOL'12*, 2012.

[12] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *AOSD'11*, pages 253–264, 2011.

[13] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A core calculus of composite layers. In *FOAL'13*, pages 7–12, 2013.

[14] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Introducing composite layers in EventCJ. *IPSJ Transactions on Programming*, 6(1):1–8, 2013.

[15] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In *AOSD'04*, pages 16–25, 2004.

[16] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *AOSD'12*, 2012.

[17] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *SC'06*, volume 4089 of *LNCS*, pages 227–242, 2006.

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