Bridging Real-World Contexts and Units of Behavioral Variations by Composite Layers

Tetsuo Kamina
University of Tokyo
kamina@acm.org

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

Hidehiko Masuhara
University of Tokyo
masuhara@acm.org

ABSTRACT

This paper proposes a new linguistic construct *composite layers* and an extension of EventCJ with it. A composite layer is implicitly activated when the declared condition is met. This extension bridges the gap between contexts and units of behavioral variations that complicates programs written in COP languages. In this proposal, only atomic layers (layers that directly correspond to a context) can be explicitly controlled by linguistic operations for layer activation. Composite layers (layers that are not atomic) are declared with a proposition constructed from other layers. Examples show that the extension simplifies programs and enhances separation of concerns.

Categories and Subject Descriptors

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

General Terms

Languages

Keywords

EventCJ, Atomic and composite layers, Implicit layer activation

1. INTRODUCTION

COP (Context-oriented Programming) [12] has been extensively studied to address the complexity raised in the development of context-aware applications. Several COP languages provide a modularization mechanism called a *layer* that modularizes behaviors executable under specific contexts, and a way to dynamically switch behaviors [3, 5, 7, 13]. In this paper, we call them *layer-based COP languages*.

There are two advantages in layer-based COP languages. First, they separate context-dependent behaviors that crosscut several traditional module systems such as classes. Second, they provide linguistic mechanisms for disciplined dynamic adoption of context-dependent behaviors. For example, ContextJ [3] and JCop [5] restrict the effects of layer activation under specified control flows. EventCJ [13] provides event-based layer transitions that make it easy to check conformance to the specification represented in the state transition model. Thus, it is easy to avoid unexpected conflicts between behaviors in these languages.

Existing layer-based COP languages assume that a unit of behavioral variations (i.e. a layer) corresponds to one context (a specific state of the system and/or environment that affects system's behaviors). This assumption is observed from the fact that, in these languages, to execute context-dependent behaviors we explicitly specify layers corresponding to the context determined by external change of status or internal actions.

However, correspondence between contexts and layers is not so simple. Each layer does not correspond to single context but corresponds to combinations of union, intersection, and negation of contexts. This *gap between contexts and layers* complicates programs written in layer-based COP languages. Specifically, independent models of context changes are tangled in the layer activation code. For example, in a mobile application, a user interface may depend not only on a history of user's operations but also on the state of the executing machine (such as status of the battery), whose changes may occur independently.

To bridge this gap, this paper proposes a new linguistic construct *composite layers* and an extension of EventCJ (a COP language with the feature of event-based layer transitions) [13] with it. In this proposal, layers are classified into *atomic* layers and *composite* layers. An atomic layer directly corresponds to a context. Only atomic layers are explicitly controlled by linguistic operations for layer activation. A composite layer is declared with a proposition in which ground terms are other layer names (true when active). The layer is activated when and only when the proposition holds. In other words, there are no ways to explicitly activate composite layers.

We demonstrate how the extension simplifies the implementation of context-aware applications written in EventCJ by using two examples, a multi-tabbed Twitter client and the CJEdit program editor [4]. Both examples show that our approach successfully simplifies programs and enhances separation of concerns.

This paper is organized as follows. Section 2 introduces an motivating example to show the problem. Section 3 sketches our proposal to tackle the problem. Section 4 discusses re-

COP'12, June 11, 2012, Beijing, China

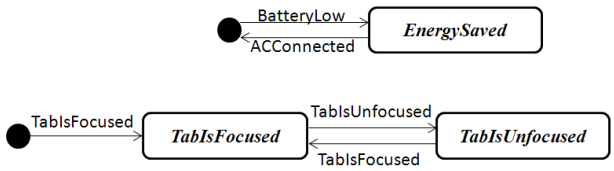

Figure 1: Context changes in the Twitter client. The black circles indicate "initial contexts" in which the system resides when it is born.

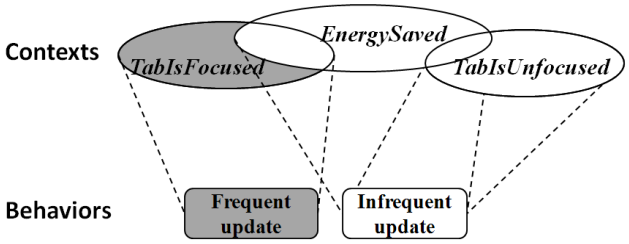

Figure 2: Correspondence between contexts and behaviors

lated work. Finally, Section 5 concludes this paper.

2. MOTIVATING EXAMPLE

In this section, we elaborate our motivation by using a multi-tabbed Twitter client. This Twitter client is equipped with multiple tabs, each of which displays the user's timeline (a list of tweets submitted by persons followed by the user), which is updated when a person followed by the user posts a tweet. At most one tab is focused at a time. A focused tab frequently updates the timeline, while other unfocused tabs infrequently update it. The user switches the focused tab by clicking. Furthermore, when the executing machine is running out of its battery, any tab (including the focused one) updates its timeline infrequently, and an alert icon is displayed.

In this application, there are two independent context changes about the focus of a tab and the machine's battery level, respectively, both of which can be discovered by applying requirements engineering methods such as the method proposed by Salifu et al. [15]. We identify three contexts, namely *TabIsFocused*, *TabIsUnfocused*, and *EnergySaved*. Context changes are modeled by using state machines, as shown in Figure 1[1].

Behaviors of the application change with respect to current contexts. There are two context-dependent behaviors: "frequent update of timeline," and "infrequent update of timeline." The correspondence between contexts and behaviors is shown in Figure 2. It shows that these behaviors depend not on single context but on a combination of multiple contexts.

2.1 Problem description

[1]We may obtain other results from the requirements elicitation process. For example, *TabIsFocused* may be considered as a default state, which makes the state transition of focus of the tab have just two states: a black circle and *TabIsUnfocused*. We do not further discuss about the requirements elicitation in this paper, but mention that our approach successfully maps contexts onto layers in both cases.

Layer-based COP languages provide a mechanism to modularize context-dependent behaviors in a layer and to dynamically switch between layers. The behavior "frequent update" in Figure 2 may be implemented in a layer, namely `TabIsActive`, and "infrequent update" may be implemented in another layer, namely `TabIsInactive`.

Layer-based COP languages dynamically activate and deactivate layers with respect to current contexts. In these languages, we *explicitly* specify layers that are activated/deactivated at particular execution points when a context changes, which implies that there is an assumption that a layer depends on single context. This mechanism complicates the program; i.e., independent models of context changes are tangled in the layer activation code. We explain this problem in the case of EventCJ [13].

In EventCJ, the execution points of layer switching are specified by events. At first, we may assume that each event corresponds to the label of edge in Figure 1 where four events are identified; two of them, namely `TabIsFocused` and `TabIsUnfocused`, change the focus of a tab, and other two, namely `BatteryLow` and `ACConnected`, change the status of battery.

However, this assumption does not hold. The problem is that these events do not directly correspond to a layer switching. For example, when `TabIsFocused` occurs, the context about focus of a tab always changes to *TabIsFocused* (Figure 1); however, `TabIsActive` becomes active only when the system is not in *EnergySaved*. Thus, in EventCJ, we have to declare `TabIsFocused` as an event that depends on the status of battery. We may declare such event by implementing a method that inspects the status of battery, namely `isBatteryLow`, and calling this method from the `if` pointcut in the event declaration:

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

In EventCJ, an event is declared with two specifications, one of which indicates when the event is generated and another one indicates where the event is sent. The former is specified using AspectJ-like pointcut sublanguage [14], and the latter is specified using the `sendTo` clause that lists instances where the event is sent. The above event `TabIsFocused` is generated when the focus of the tab is changed and sent to the selected tab. The `if` pointcut ensures that the event is generated only when the result of `isBatteryLow` is `false`.

This approach has two disadvantages. First, the program does not directly reflect the model of context changes. For example, `TabIsFocused` identified in Figure 1 and the above event declaration are different; while the former is generated whenever the tab is getting focused, the latter is generated only when the result of `isBatteryLow` is `false`. In other words, two context changes about the focus of the tab and the status of battery are tangled in the same event declaration. Second, this approach complicates the layer activation code. For example, when `ACConnected` in Figure 1 is generated, in EventCJ there should be two different layer transition rules: one for when the tab is in *TabIsFocused*, and the other for when the tab is in the initial state. Thus, we need to declare two different events for each case, and to

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

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

transition BatteryLevelLow: -> EnergySaved;

transition ACConnected: EnergySaved ->;
```

Figure 3: Context transition rules in the extension

develop two different layer transition rules for each event.

In other words, there is a gap between contexts and layers. In EventCJ, every switching of layers is explicitly controlled by layer transition rules. However, context changes do not always trigger switching of layers.

3. OUR PROPOSAL

To tackle the aforementioned problem, we propose a new linguistic construct called *composite layers*. A composite layer depends on activation of other layers. It declares a proposition in which ground terms are other layer names (true when active), and is activated when and only when the proposition holds. We also propose an extension of EventCJ with composite layers.

In this paper, layers that are not composite are called atomic layers. Only atomic layers can be activated explicitly (by using layer transition rules).

3.1 Atomic layers

An atomic layer is a layer that directly corresponds to a context. It is declared with existing syntax of layer declarations. If there is a one-to-one correspondence between contexts and behaviors, we implement such behaviors in atomic layers using partial methods. In the Twitter example, there are no such correspondence; thus, we declare each context as an atomic layer with an empty body. In the most of layer-based COP languages, such layers are declares as follows:

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

In the extension of EventCJ, only atomic layers are controlled by layer transition rules. Since there is a one-to-one correspondence between contexts and atomic layers, layer transition rules can be mechanically derived from the state transition model of context changes. For example, from state transition models shown in Figure 1, we derive layer transition rules shown in Figure 3. The syntax is the same as that is shown in [1]. For example, the first transition rule is read as, "upon the generation of `TabIsFocused`, if `TabIsUnfocused` is active, then it becomes inactive and `TabIsFocused` becomes active; otherwise, only `TabIsFocused` becomes active."

3.2 Composite layers

A composite layer is used to modularize behaviors that depend on a combination of multiple contexts. It extends the syntax of layer declarations; a composite layer is declared with a condition that specifies when the layer is active:

```
layer TabIsActive
    when TabIsFocused && !EnergySaved {
  /* frequent update of timeline */
}
layer TabIsInactive
    when TabIsUnfocused || EnergySaved {
  /* infrequent update of timeline */
}
```

The `when` clause can refer to other layers to specify when it is active. Each referred layer is interpreted as a proposition that is true when it is active. Thus, the first layer declaration shown above is read as, "`TabIsActive` is active when `TabIsFocused` is active and `EnergySaved` is not active." Similarly, the second layer declaration is read as, "`TabIsInactive` is active when `TabIsUnfocused` or `EnergySaved` is active."

This introduction of `when` clause raises discussion about the style of layer declarations. Most of the layer-based COP languages employ the so called *layer-in-class* style [2], which means that layers are declared within classes. This style is not suitable for this extension. In the layer-in-class style, layer declarations for the same layer are distributed among classes that contain that layer. Thus, the addition of the `when` clause requires that there should be some mechanisms to ensure the consistency of layer declarations and/or programmers have to write the same condition for every layer declaration distributed among classes. The *class-in-layer* style in which (partial definitions of) classes are declared within a layer may avoid such inconvenience.

The aforementioned problem is tackled by the proposed extension. The context transition rules shown in Figure 3 is straightforwardly derived from the state machines in Figure 1. While designing and implementing events and context transition rules about the status of the tab, we do not have to consider the status of battery. Thus, the proposal enhances separation of concerns. Furthermore, our approach reduces the numbers of event declarations (i.e., we do not have to declare multiple `ACConnected` events for each status of the tab), which also reduces lines of layer transition rules.

3.3 Another example: CJEdit [4]

To verify the proposed approach is effective in other applications, we introduce another example CJEdit, a program editor providing different functionalities with respect to the cursor position [4]. CJEdit is a program editor that enhances the readability of programs by providing different text formatting techniques for code and comments. The code part is formatted in a typewriter format with syntax highlighting, and the comment part is formatted in a rich text format (RTF) that supports multiple fonts, text sizes, decorations, and alignments. Furthermore, CJEdit provides different GUI components depending on whether the programmer writes code or comments.

There are two context changes about cursor position and text region, respectively. The former has two states, namely *CursorOnCode* and *CursorOnComments*, and the latter has two states, namely *RenderingCode* and *RenderingComments*. Figure 4 shows state transition models for both context changes[2].

[2]Having default states (i.e., the states that are not *Cur-*

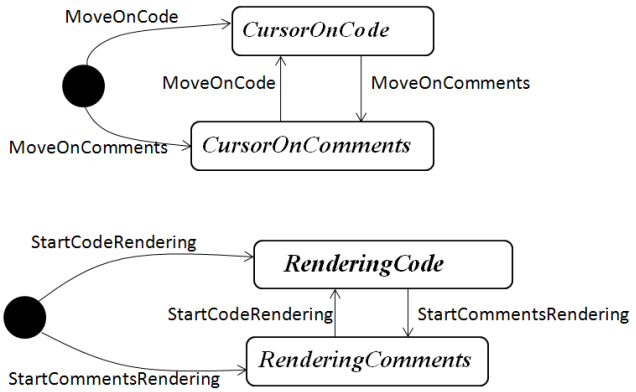

Figure 4: Context transition system in CJEdit

There are five context-dependent behaviors that change with respect to current contexts. Each of them can be modularized by layers. We list these behaviors with layer names as follows:

- `CursorOnCode`: showing the GUI components for code editing functionalities such as outline view of the program structure.
- `CursorOnComments`: showing the GUI components for RTF functionalities such as menu items and tools for fonts, text sizes, decorations, and alignments.
- `RenderWithHighlighting`: rendering the program text in typewriter format with syntax highlighting.
- `RenderWithoutHighlighting`: rendering the program text in typewriter format without syntax highlighting.
- `RenderingComments`: rendering comments in RTF.

We summarize correspondence between contexts and behaviors in Figure 5. While the former three behaviors directly correspond to contexts *CursorOnCode*, *CursorOnComments*, and *RenderingComments*, respectively, the latter two do not; `RenderWithHighlighting` depends on two contexts *CursorOnCode* and *RenderingCode*, and `RenderWithoutHighlighting` depends on *CursorOnComments* and *RenderingCode*.

We can implement CJEdit by EventCJ with composite layers. The former three behaviors that have one-to-one correspondence to contexts can be implemented by the following atomic layers:

```
layer CursorOnCode {
 /* code editing functionalities */
}
layer CursorOnComments {
 /* comments editing functionalities */
}
layer RenderingComments {
 /* rich text formatting */
}
```

sorOnCode and *CursorOnComments*, and not *RenderingCode* and *RenderingComments*, respectively) in the model is a precise interpretation of the original implementation of CJEdit, which is not in any contexts at the very beginning of its startup.

```
transition MoveOnCode:
  CursorOnComments ?
    CursorOnComments -> CursorOnCode
| -> CursorOnCode;

transition MoveOnComments:
  CursorOnCode ?
    CursorOnCode -> CursorOnComments
| -> CursorOnComments;

transition StartCodeRendering
  RenderingComments ?
    RenderingComments -> RenderingCode
| -> RenderingCode;

transition StartCommentRendering
  RenderingCode ?
    RenderingCode -> RenderingComments
| -> RenderingComments;
```

Figure 6: Layer transition rules for CJEdit by our proposal

For context *RenderingCode*, there are no one-to-one relations to any behaviors. Thus, we declare it as an atomic layer with an empty body:

```
layer RenderingCode {}
```

Using atomic layers, we implement the remaining behaviors as composite layers:

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

Figure 6 shows layer transition rules that are simply derived from state machines shown in Figure 4. Note that, as in the Twitter example, two independent models of context changes (that will be tangled in the original EventCJ) are separated in the layer transition rules.

4. RELATED WORK

Most of the existing COP languages manage layer activation *per-thread* manner by using the block structure (called `with`-block) [3, 5, 7, 11]. This style of layer activation makes it difficult to develop applications in which context-dependent behaviors inherently occur per-instance, like the Twitter client shown in this paper.

The tangling problem of layer activation occurs in those COP languages, and thus the proposed approach may also effective in them. For example, in ContextJ [3] we need to use a workaround using first-class layers to store effective layers in a variable, and activate the stored layers whenever a context-dependent behavior is needed. For example, in CJEdit we switch active layers in the following method in which a context change occurs:

```
void onCursorPositionChanged() {
```

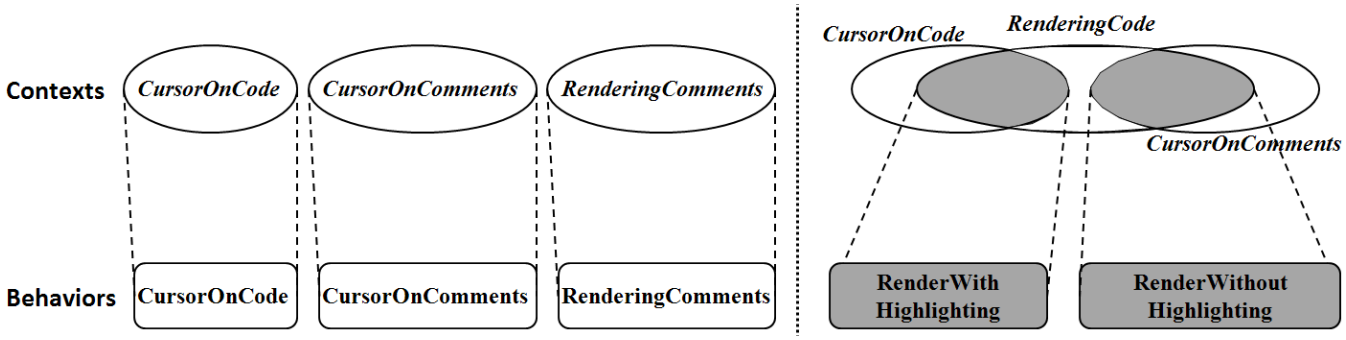

Figure 5: Correspondence between contexts and behaviors in CJEdit

```
  if (Layer.isActive(CursorOnComments)) {
    /* Deactivate CursorOnComments */
    /* Activate CursorOnCode */
    /* Other managements for highlighting.. */
  } else { .. }
}
```

We need to manage layer switching with considering not only cursor position but also text highlighting, whose changes occur independently and should be separately described. Implicit activation of composite layers addresses this problem.

Ambience [10] and its successor AmOS [9] are prototype-based context-oriented languages featuring multimethods and subjective dispatch. Unlike layer-based COP languages, a context is reified as an object that is implicitly argumented to the method invocation. Thus, each context-dependent method is defined with a context, which can be a combined context that is similar to a composite layer. While a composite layer is declared with an arbitrary proposition, Ambience and AmOS only support intersections.

Subjective-C [8] is an extension of Objective-C with context-orientation concepts. Like Ambience and AmOS, in Subjective-C, context-dependent behaviors are defined for each method using the `#context` annotation that specifies a context on which the method depends. It provides a small domain-specific language (DSL) to represent relations between contexts.

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.

Costanza and D'Hondt proposed a method to analyze the dependency between layers using feature diagrams [6], where each feature is mapped onto a layer. They provide an extension of ContextL [7] to represent composite layers (layers that correspond to composite features). Since their proposal can represent relations between layers like "layer `A` includes one of layers `B`, `C`, and `D`" and/or "layer `X` includes all layers `Y` and `Z`," it shares similarities with our approach. The difference is that their approach provides direct activation of composite layers, where other layers depending on activated layers are automatically activated. In contrast, our proposal activates all composite layers implicitly and all atomic layers explicitly.

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.

5. CONCLUDING REMARKS

In this paper, we presented an extension of EventCJ in which layers are classified into atomic and composite layers. Contexts determined by external change of status or internal actions are directly mapped onto atomic layers, and thus context changes are directly represented by layer transition rules. A layer that depends on other layers is declared as a composite layer that specifies a condition about when the layer becomes active. All composite layers are implicitly activated when the condition becomes true. This proposal addresses the problem of the gap between contexts and layers, which suffers most of the existing layer-based COP languages.

As future work, we are planning to formally define operational semantics of this extension, and to implement an efficient compiler.

6. REFERENCES

[1] 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.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. 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*, 28(1):272–292, 2011.

[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, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Proceedings of the*

International Conference on Software Composition 2010 (SC'10), volume 6144 of *LNCS*, pages 50–65, 2010.

[6] Pascal Costanza and Theo D'Hondt. Feature descriptions for context-oriented programming. In *2nd International Workshop on Dynamic Software Product Lines (DSPL'08)*, 2008.

[7] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *Dynamic Language Symposium (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] Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *DLS'07*, pages 77–88, 2007.

[11] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *GTTSE 2007*, volume 5235 of *LNCS*, pages 396–407, 2008.

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

[13] 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.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Grisword. An overview of AspectJ. In *ECOOP'01*, pages 327–353, 2001.

[15] Mohammed Salifu, Yujun Yu, and Bashar Nuseibeh. Specifying monitoring and switching problems in context. In *RE'07*, pages 211–220, 2007.

[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 2006*, 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: Proceedings of the 2007 international conference on Dynamic languages*, pages 143–156, 2007.