Designing Event-based Context Transition in Context-oriented Programming

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 programming language EventCJ. Its design stems from our observation that, in many context-aware applications, context changes are triggered by external events. Thus, in addition to the current COP language mechanisms, namely the one to activate/deactivate layers in accordance with a flow of control in programs, and the one to dispatch method calls to partial methods on active layers, we propose a mechanism to declaratively switch contexts of the receiver of events. EventCJ can declare events that trigger context transitions, and context transition rules that define how each instance's context changes when it receives a specific event. After the transition, the instance acquires the context dependent behaviors provided by the activated context. Each event is declared in an AspectJ-like pointcut that specifies where the event is fired in the join points of the system. EventCJ separates the specification of when each context is activated and deactivated that may crosscut whole program in the existing COP languages. Furthermore, the declarative nature of the context transition rules help validation of some properties that the contexts should satisfy.

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

ContextJ, Context translation rules, EventCJ

1. INTRODUCTION

COP'10, June 22, 2010, Maribor, Slovenia

Context awareness, which is a capability to behave with respect to its surrounding context, is becoming a major concern in many application areas such as ubiquitous computing, adaptive user interfaces, and business applications. In such applications, a behavior of the system may change with respect to its surrounding context. Thus, identification and a dynamic composition mechanism of contexts are the fundamental requirements to develop such applications. Moreover, such dynamic composition increases the complexity of software, thus it is desirable to implement the variation of behaviors in a modular way, and composition of such variations should be explicitly controlled to make it easy to reason about some required properties.

Context-oriented programming (COP) [6, 8, 9] has been proposed to modularly implement context-aware applications. In COP, context dependent behavioral variations can be modularized as layers of partial methods, and these variations can be dynamically activated and deactivated at runtime. Existing COP languages have two linguistic mechanisms, namely a mechanism to activate/deactivate layers, and a mechanism to dispatch method calls to partial methods on active layers. However, in many COP languages, it is still not easy to specify *when* a context should be activated or deactivated.

As Appeltauer et al.[3] pointed out, some context activations/deactivations occur upon external events. However, most current COP languages control context activation by using block statements. Although block statements are convenient to represent when a context in the problem domain is active during particular method calls in the implementation, they are not suitable to implement context activations that are triggered by events (such as entering a building from out of doors, or moving a cursor from a GUI component to another), which can occur at any time during the execution of a program. Therefore, in such languages, programmers have to declare an event handler for each event and write context activation code that may crosscut whole the program. Furthermore, since a context activation is specified by a block statement, its effect cannot extend into the caller. Thus, after returning from the event handler, the activated context is deactivated and the programmers have to write a boilerplate code that remembers the current activated contexts and activates them again when they are required.

One of the possible solutions to address this problem is to exploit advice mechanism in aspect-oriented programming (AOP) [12] for localizing code that globally activates and deactivates contexts. For example, we can use the AspectJ-like pointcut and advice to specify where context activation

is performed [13]. Within an advice, we can provide imperative statements that activate contexts. However, such imperative statements make it very difficult to reason about properties like "the context L will not be activated during the method m is executed."

All in all, in current COP languages, context activation is controlled *per thread* and scoped to the dynamic extent of a block of statements. However, to reactively switch contexts by external events, we require an additional mechanism to declaratively specify how the receiver of events react to those events, namely rules for context activation/deactivation and routines associated with events.

We propose a new programming language EventCJ that allows programmers to control contexts *per instance* by declaring *events* and *context transition rules*: an event triggers context transitions and a context transition rule defines how contexts of each object are changed in responce to events. Each event is specified by using a pointcut-like language in AspectJ. Context transition rules separate the specification on context transitions from the program. Furthermore, they help validation of some properties that the contexts should satisfy.

This paper only shows our ideas and preliminary design of EventCJ. It also shows a case study of a banking account application and provides a comparison among other related work and future direction of our research.

2. MOTIVATION

2.1 A Simple Example

This section describes our motivation of developing a new programming language that supports event-based context transitions by using a simple example. This example features a pedestrian navigation system that behaves differently with respect to its surrounding environment. If it is used out of doors, it provides a GPS based navigation for the user. If it is used inside a building, it receives signals from active RFIDs to identify the location within the building. To save energy, the RFID reader is automatically switched off outside the building, and the GPS receiver is automatically switched off inside the building.

By using COP languages, we can modularize the context-dependent behavioral variations. Figure 1 shows a piece of pedestrian navigation system written in ContextJ [2], a COP language based on Java. We declare GPS navigation specific behaviors within a *layer* `GPSNavi`, and RF navigation specific behaviors within a layer `RFNavi` (throughout this paper, we use the words "layer" and "context" interchangeably). For example, in Figure 1, the `DeviceController` class declares three methods, namely `powerOffDevices()`, `startDevices()`, and `currentStatus()`, and the layers `GPSNavi` and `RFNavi` declare partial methods that provide behavioral variations for these methods. Examples of such behavioral variations are as follows: when the layer `GPSNavi` is activated, the `startDevices()` method starts the GPS receiver, or when the layer `RFNavi` is activated, the `powerOffDevices()` method turns the RFID reader off, and so on.

In ContextJ, the context-dependent behaviors can dynamically be composed with the base program by using the `with` statement. For example, when we receive an RF event that indicates that we enter the building, the following event handler may be invoked:

```
1  class DeviceController{
2    void powerOffDevices(){...}
3    void startDevices(){...}
4    void currentStatus(){...}
5    layer GPSNavi{
6      after void powerOffDevices(){...}
7      after void startDevices(){...}
8      after void currentStatus(){...}
9    }
10   layer RFNavi{
11     after void powerOffDevices(){...}
12     after void startDevices(){...}
13     after void currentStatus(){...}
14   }
15 }
```

Figure 1: Example of context dependent behaviors in ContextJ

```
1 void onBuildingEntered(RFEvent e){
2   with(GPSNavi){ powerOffDevices(); }
3   with(RFNavi){ startDevices(); }
4 }
```

This event handler firstly turns the GPS receiver off (because the `powerOffDevices()` method is called within the `GPSNavi` layer), then starts the RF reader (because the `startDevices()` method is called within the layer `RFNavi`).

2.2 A Problem

The `with` statement of ContextJ is convenient to explicitly activate each layer within a specific control flow. However, it is not suitable to implement context transitions that are triggered by external events, which may occur at any time. Therefore, in the above example, the `with` statement has to always be used within event handlers that capture the relevant events. When the event handler returns, the activated layer is also deactivated, because the execution proceeds outside the `with` statement. Therefore, there are no ways to get the layer specific `currentStatus()` after returning from the event handler, unless we provide a wrapper method that is used to inspect the current machine's status:

```
1 void wrapCurrentStatus() {
2   with(lastLayer()){ currentStatus(); }
3 }
```

This method firstly retrieves the layer that was active last time, then activates it by using the `with` statement and calls the layer specific `currentStatus()` method. In this case, we have to provide another boilerplate code; we have to remember the layer that was active last time and retrieve it by using the auxiliary `lastLayer()` method.

The source of this problem is, in ContextJ, layer activation triggered by events cannot directly be captured by using the `with` statement and thus we have to use it within the event handlers on a case-by-case basis, and there are no ways to represent the layer activation that extends into executions after returning the method where the layer is activated.

2.3 An AOP-based Solution and Its Problem

A possible solution to switch context is to use an AspectJ-like pointcut and low-level context activation/deactivation

```
pointcut RFEvent(): execution(
  void DeciveController.onBuildingEntered);
before(Object o): RFEvent() && this(o)
       && if(Layer.isActive(GPSNavi)){
  powerOffDevices();
  Composition.deactivateLayer(GPSNavi);
  Composition.activeLayer(RFNavi);
  startDevices();
}
```

Figure 2: Example of context transition in AspectJ

```
class DeviceController{
  declare event RFEvent(DeviceController dc):
    before execution(
      void DeviceController.onBuildingEntered()
  ) && this(dc) : receiver(dc);
  event RFEvent: GPSNavi -> RFNavi
    before { powerOffDevices(); }
    after { startDevices(); }
  ..
}
```

Figure 3: `event` declaration in our proposal

primitives (such as ContextJ reflection API) to declare an advice that makes context transitions. We consider that, after the transition, the context activation lasts until the next event that deactivate the activated context is fired, which can also be captured by using a pointcut.

Figure 2 shows the pedestrian navigation example written in AspectJ. The `RFEvent()` pointcut captures the execution of `onBuildingEntered()` method declared in `DeviceController`. The advise is executed before the `onBuildingEngtered()` method is executed. It firstly checks whether the `GPSNavi` layer is active or not by using the `isActive()` method provided by the ContextJ reflextion API; if so, then it turns the GPS receiver off, deactivate `GPSNavi`, activate `RFNavi`, and starts the RFID reader.

However, these imperative primitives for context activation/deactivation make it difficult to reason about some required properties imposed by some context transition specifications. For example, someone would like to statically ensure that the RFID reader always inactive in the out of doors (e.g., by using some tools); however, ensuring it is quite difficult in this AOP-based approach.

3. OUR PROPOSAL

This section proposes a new programming language EventCJ to address the aforementioned problems. To represent event-based context transitions, EventCJ provides a new language construct named *context transition specification* that denotes context transition rules and before- and after-procedures that are performed before and after the transition occurs, respectively. As in ContextJ, each context is declared by using the `layer` declaration statement placed within a class declaration. Thus, currently we take the layer-in-class style[1]. To explain the constructs of EventCJ, we use the same example of section 2. The structure of layer declaration is identical to that of ContextJ (shown in Figure 1), thus in this section, we omit this part.

In EventCJ, a context transition is triggered by an event. An event is declared by using the `declare event` statement with a name of event and a specification that specifies when this event will be fired by using the AspectJ-like pointcut. For example, in Figure 3, `RFEvent` is declared to be fired at an execution of method `onBuildingEntered()` declared in `DeviceController`. The receiver of `RFEvent` is specified by the `receiver` clause; in this case, the receiver of `RFEvent` is the instance itself that executes the `onBuildingEntered()` method. In the receiver clause, we may specify a set of instances by listing multiple instances, or by using the connotative representation as follows:

```
receiver(DeviceController dc | dc.id > 0 )
```

If the `receiver` clause is not provided, the effect of context transition is global.

Figure 3 also describes an example of a context transition specification that specifies how the receiver of events switches its active contexts. The specification starts from the keyword `event`, followed by a sequence of event names. In Figure 3, this sequence of events is specified as a single event `RFEvent`, but we can also specify a pattern of events.

This sequence of event names is followed by a *context transition rule* that specifies how the context of the receiver of events changes with respect to the received events. For example, the context transition rule `GPSNavi -> RFNavi` described in Figure 3 specifies that if `GPSNavi` exists in the set of activated contexts, then the `GPSNavi` context is deactivated and the `RFNavi` context is activated. The left-hand side of `->` is a *condition* that restricts when the context transition can occur. In the above example, the context transition to `RFNavi` will not occur unless `GPSNavi` is activated.

Besides the operator `->`, we can also use a context transition operator `+>` that indicates the context of the left-hand side will *not* be deactivated and the context of the right-hand side will be activated. We can use a notation `*` to represent a (possibly empty) set of currently activated contexts. Furthermore, we can use a notation `.` to represent "nothing." For example, we can encode the execution that all the activated contexts are deactivated when the navigation system is turned into manual mode:

```
event SetManual: * -> . ..
```

We can create a composite context transition rule by concatenating each context transition rule by using `||` (the first matched context transition rule is selected) and `&&` (all the context transition rules are selected iff all the rules satisfy the conditions). For example, the following context transition rule specifies that if `GPSNavi` is active in the receiver of `RFEvent`, it is deactivated and `RFNavi` becomes active; otherwise, if `RFNavi` is active, it is deactivated and `GPSNavi` becomes active:

```
event RFEvent: GPSNavi -> RFNavi
  || RFNavi -> GPSNavi ..
```

We can also restrict the condition that specifies when the context transition occurs. For example, in Figure 3, `RFNavi`

[1]This "current" decision may be reviewed in the future, which is not in the scope of this paper.

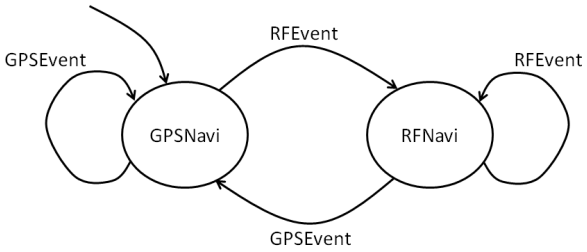

Figure 4: Example of context transitions in the pedestrian navigation system

will be activated whenever an instance of `DeviceController` receives an event `RFEvent` and `GPSNavi` is active, but we can also represent an exclusive match (i.e., the condition where `GPSNavi` is active *and no other contexts are active*) as follow:

```
event RFEvent: GPSNavi! -> RFNavi ..
```

Furthermore, we can specify the *not* condition. For example, we can declare a rule specifying that if `RFNavi` is not active, then it will be activated:

```
event RFEvent: !RFNavi +> RFNavi ..
```

Finally, we can specify the procedures that is executed *before* and *after* the transition occurs by using the `before` block and the `after` block, respectively. For example, in Figure 3, `powerOffDevices()` is executed before the context transition and `startDevices()` is executed after the context transition. Note that, before the context transition, `GPSNavi` is still active and `RFNavi` is not active yet. Thus, the execution of `powerOffDevices()` results in the execution where the device controller turns the GPS receiver off.

By EventCJ, we can declaratively specify which context is activated by which event, and this activation is preserved until the next event that deactivate the context is fired. The specification that specifies when each context is (de)activated is thus separated from the program. We do not have to write any auxiliary context management code (such as storing the context that was active last time and retrieving it) that can easily be scattered into the program written in ContextJ.

Unlike the pure AOP approach sketched in section 2.3, in EventCJ, where the layer activation occurs is restricted by the pointcut. Even though there may be some consistency issues (unlike the `with` block statement, there may be some possibilities that an event that triggers the activation of certain layer occurs during a certain action), this restriction may help us to avoid such inconsistency by analyzing code (possibly by using some tools). Furthermore, declaration of context transition rules helps validation of some required properties that the contexts should satisfy. For example, from the context transition rules we can form a state machine shown in Figure 4. This state machine ensures that the contexts `GPSNavi` and `RFNavi` will not become active at the same time, and provided that the `GPSEvent` event will not be fired inside the building and the `RFEvent` event will not be fired outside the building, it can easily be ensured that `RFNavi` will not become active outside the building.

4. CASE STUDY

This section shows that EventCJ is expressive enough to implement applications straightforwardly with respect to

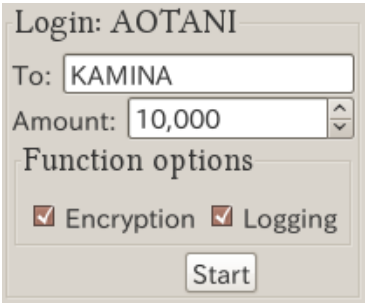

Figure 5: Client of the money transfer system

```
1  class TransferSystem{
2    void transfer(Account from, Account to, int am){
3      from.debit(am);
4      to.credit(am);
5    }
6    layer Encryption{
7      void transfer(Account from, Account to, int am){
8        proceed(from, to, encrypt(am));
9      }
10   }
11   layer Logging{
12     after void transfer(Account from,
13                         Account to, int am){
14       Logger.logTransfer(from, to, am);
15     }
16   }
17   /* other methods and constructors */
18 }
```

Figure 6: The class `TransferSystem` in the money transfer system

their specifications by using a simple money transfer system [2] that handles the transfer of an amount of money from one bank account to another.

Figure 5 shows the GUI part of the system. Here the user AOTANI logs in and transfers 10,000 yens from his account to KAMINA's account. When the start button is pressed, it starts the money transfer.

The two check boxes "Encryption" and "Logging" control the encryption and logging functions, respectively. If the check box "Encryption" is checked, the system encrypts the information about the amount of transferred money. If the check box "Logging" is checked, the system logs a sequence of operations executed during the money transfer: i.e., debiting 10,000 yens from AOTANI and crediting 10,000 yens to KAMINA.

The implementation of the money transfer system in EventCJ is straightforward. Figures 6 and 7 show the `TransferSystem` and `Account` classes that implement the core parts of the system. The behaviours of encryption and logging functions are implemented modularly as layers, namely `Encryption` and `Logging`.

Each layer is activated and deactivated when the state of the corresponding check box is changed. This can be achieved straightforwardly by (1) declaring an execution of the methods invoked when the check boxes change their

```
class Account{
  void credit(int am){...}
  void debit(int am){...}
  layer Encryption{
    void credit(int am){
      proceed(decrypt(amount));
    }
    void debit(int am){
      proceed(decrypt(amount));
    }
  }
  layer Logging{
    after void credit(int am){
      Logger.logCredit(this, am);
    }
    after void debit(int am){
      Logger.logDebit(this, am);
    }
  }
  /* other methods and constructors */
}
```

Figure 7: The class `Account` in the money transfer system

states as an event and (2) changing the states (active or inactive) of the layers when the event occur:

```
1  /* activating/deactivating the encryption layer*/
2  declare event SwitchEncryption: before execution(
3    /*handler for changes of the encryption check box*/);
4  event SwitchEncryption:
5    !Encryption +> Encryption //activating Encryption
6    || Encryption -> .        //deactivating Encryption
7    ;                         //no procedures is executed

9  /* activating/deactivating the logging layer*/
10 declare event SwitchLogging: before execution(
11   /*handler for changes of the logging check box*/);
12 event SwitchLogging:
13   !Logging +> Logging       //activating Logging
14   || Logging -> .           //deactivating Logging
15   ;                         //no procedures is executed
```

Lines 2–7 declare an event and context transition rules regarding the `Encryption` layer. Lines 2–3 declare the event `SwitchEncryption` and specify that it is fired before the execution of the handler for the state change events of the encryption check box. When `SwitchEncryption` occurs, the layer `Encryption` is activated if it is not active and is deactivated otherwise. Neither before nor after procedures are specified. Lines 10–15 declare an event and context transition rules of the `Logging` layer in the same way.

5. RELATED WORK

JCop [4] is a successor of ContextJ that supports declarative statements specifying where each layer to be active with respect to control flows by using an AspectJ-like pointcut syntax. Although JCop is developed independently of EventCJ, both share the same motivation. JCop can separate the specification of event specific context activation from the program, thus it enables modular implementation of context dependent features in event-based applications such as GUI applications. However, it does not support EventCJ-like declarative statements for context transition rules and procedures associated with context transitions. Thus, validation of some properties that the context should be satisfy would be rather difficult in JCop.

There are some programming languages designed for event-based programming. EventJava [7] is an extension of Java that seamlessly integrates events with methods and broadcasting with unicasting of events. In EventJava, each event is declared and invoked as a method, but broadcasting of events (syntactically like a static method call) and event correlations (specifying the pattern of events to react) are supported. Each event conveys attributes that forms a context, and this context is customizable [10], but this context is global and only one context is supported per application. ECaesarJ [14], an extension of CaesarJ [5], provides a mechanism to declare events and their handlers. By using them, we can encode context transitions like that explained in this paper and their associated procedures. However, in ECaesarJ, the context activation does not cause the change to the context dependent behaviors that is supported in EventCJ. Furthermore, ECaesarJ can capture only external events that are explicitly fired by the source of events, whereas in EventCJ, implicit internal events can also be captured by using pointcuts.

CSLogicAJ [16] is a programming language features the adaptation of service behavior by context-sensitive service aspects. It is based on LogicAJ [15] that is an aspect-oriented extension of Java supporting metavariables in pointcuts. As in EventCJ, context change is modeled in terms of join points of the system. CSLogicAJ is based on the OSGi-based middleware, thus only services available in the local OSGi registry can be intercepted.

Context activation per instance was also proposed in NextEJ [11], which extends the `with` block statement to specify the instances that are affected by the context activation within a specific control flow. Unlike EventCJ, the context activation semantics of NextEJ is rather similar with that of ContextJ (the scope of context activation is controlled by the `with` block statement), but NextEJ also provides the morphing of instance types (the type of instance can be refined within the block statement so that it acquires new behaviors). Realizing such morphing is very hard in EventCJ. Since context activation of NextEJ is based on the `with` block statements, it is not suitable to represent event-based context transitions.

Languages for trace/event-based AOP [1, 17] use pointcuts similar to EventCJ. Although current EventCJ does not observe sequences of events but only one event, it might be one option in future versions to use such a language to handle sequences of events if that is necessary and useful to represent context transitions.

6. CONCLUSION AND FUTURE WORK

The event-based context transition in EventCJ makes it possible to declaratively specify how the set of active contexts associated with an instance changes with respect to its receiving events. Events are declared with pointcuts that specify where each event is fired in the join points of the system, thus we can flexibly specify the execution points where context transition occurs responding external events. By the `layer` construct taken from ContextJ, we can im-

plement context dependent behaviors, but the activation of these behaviors now can extend over multiple methods. The state machine like notation makes it easier to ensure some required properties are satisfied in the program. The case study of banking account application expects its applicability to business applications.

However, this work is still in its early stage. One of the planned future work is to implement a compiler and re-implement the existing real applications to evaluate the effectiveness of EventCJ. As mentioned above, context transition rules help validation of some properties that the contexts should satisfy. Thus, applying automated checking methods (such as model checking) to the program written in EventCJ will also be an important direction of our research.

7. REFERENCES

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of OOPSLA'05*, pages 345–364, 2005.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. In *Proceedings of the JSSST Annual Conference 2009*, 2009.

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

[4] 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)*, 2010. to appear.

[5] Ivia Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173, 2006.

[6] 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.

[7] Patrick Eugster and K.R. Jayaran. EventJava: An extension of Java for event correlation. In *ECOOP'09*, volume 5653 of *LNCS*, pages 570–594, 2009.

[8] 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.

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

[10] K.R. Jayaran and Patrick Eugster. Context-oriented programming with EventJava. In *COP'09*, 2009.

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

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, 1997.

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

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

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

[16] Tobias Rho, Mark Schmatz, and Armin B. Cremers. Towards context-sensitive service aspects. In *Proceedings of the Workshop on Object Technology for Ambient Intelligence and Pervasive Computing, collocated with ECOOP'06*, 2006.

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