Context-Aware Aspects

Éric Tanter[1] Kris Gybels[2] Marcus Denker[3] Alexandre Bergel[4]

[1]Center for Web Research/DCC
University of Chile, Santiago – Chile

[2]PROG Lab
Vrije Universiteit Brussel – Belgium

[3]Software Composition Group
University of Bern – Switzerland

[4]Distributed Systems Group
Trinity College – Ireland

Abstract. Context-aware applications behave differently depending on the context in which they are running. Since context-specific behavior tends to crosscut base programs, it can advantageously be implemented as aspects. This leads to the notion of *context-aware aspects*, *i.e.*, aspects whose behavior depends on context. This paper analyzes the issue of appropriate support from the aspect language to both restrict the scope of aspects according to the context and allow aspect definitions to access information associated to the context. We propose an open framework for context-aware aspects that allows for the definition of first-class contexts and supports the definition of context awareness constructs for aspects, including the ability to refer to past contexts, and to provide domain- and application-specific constructs.

1 Introduction

Context awareness [4, 9], *i.e.*, the ability of a program to behave differently depending on the context in which it is running, has been the subject of a number of research proposals, mainly in the field of ubiquitous computing [21], self-adaptive [18] and autonomic systems [17]. In these areas, a major issue is that of *perceiving* the context surrounding an application (*e.g.*, hardware or network state, user characteristics, location, etc.). Context awareness toolkits have been proposed to address this issue of context perception (*e.g.*, WildCAT [8]). In the area of programming languages, it has been recognized that beyond perceiving context, actually *composing* context-specific behavior with the application logic results in context-related conditionals (`if` statements) being spread out all over the program [6]. Context awareness is therefore a crosscutting concern, which is a good candidate for being treated as an aspect. Doing so implies that aspects should be context aware; a *context-aware aspect* is an aspect whose behavior depends on the context. The "behavior" of an aspect includes the crosscut specification of the aspect as well as its associated action (*a.k.a.*, advice). This paper explores aspect *language constructs* to scope aspects to certain contexts, and to allow aspect advices to be parameterized by information associated to contexts.

Current approaches to AOP support several means to restrict an aspect based on some kind of *context*: *e.g.*, based on certain data associated with the join

points it intercepts (*e.g.*, the value of a certain parameter), or based on their relationship to past join points (*e.g.*, current control flow or past execution history [11, 13]). However the notion of context used in context-aware applications is more general and AOP languages lack an explicit notion of context. Interestingly, context can also be related to particular application domains, such as a promotional context in an online shopping application. Also, current AOP languages are limited with respect to the kind of context dependencies that can be expressed: most do not consider *past* contexts. While there are approaches that keep track of past events and state of the program, the problem of context-dependent aspects has not been fully considered in these cases. Expressing context dependencies in aspects would imply relying on design patterns and idioms. Conversely, we believe context awareness is central enough to many systems to deserve dedicated language constructs. We therefore aim at studying appropriate aspect language constructs for context-aware aspects, including general-purpose and domain-/application-specific constructs. The contributions of this paper are: *(a)* a general analysis of the issues associated with context-aware aspects; *(b)* the description and implementation of an open framework for context-aware aspects that meets all identified requirements. This framework, implemented on the Reflex AOP kernel [14], includes a context definition framework and a framework for defining context-related aspect language constructs.

Section 2 introduces a running example on which we base our analysis of context-aware aspects, and distills a number of requirements for defining contexts and providing aspect language features for context awareness. We describe our framework for context-aware aspects in Section 3, and discuss related work in Section 4. Section 5 presents our conclusions.

2 Motivation and Requirements

2.1 Running Example

The running example is an online shop application. When a customer logs in, a shopping cart is created that can be filled with various items. When the purchase has to be ordered, the bill is calculated. Within this application, we consider a simple discounting aspect. The following is a skeleton implementation of a Discount aspect in AspectJ which is not at all dependent on current promotions but simply applies a constant discount of 10%:

```
aspect Discount {
  double rate = 0.10;
  pointcut amount() : execution(double ShoppingCart.getAmount());
  double around() : amount() { return proceed() * (1 - rate); }
}
```

There are several ways in which the discounting aspect can be related to a promotion. For instance, the discount can be based on **(D1)** the current promotion that is active either when the customer checks out, or **(D2)** when the customer

logs in and the shopping cart is created, or **(D3)** when the item is added to the shopping cart.

The fact that a promotion is active or not can be described as the application being in a *promotional context* [1]. Whether the application is currently in a promotional context can be defined in several ways, e.g.: **(P1)** it can be time-based, based on whether the current time falls in any of the given intervals during which promotions are given; or **(P2)** the shop application can automatically give promotions when the shop's stock area is getting overloaded and needs to be cleared; or **(P3)** the shop's sales department may want to advertise a new web-services-based interface to the application, the application would then be in a promotional context if shopping is done through web-service requests.

Finally, the discount rate may **(R1)** simply be constant as in the example above, but **(R2)** may also vary depending on the actual promotion. In other words, instead of being in a promotional context or not, the application may also offer promotions with different discount rates.

2.2 Example Design Analysis

Context Definition. The definition of when to apply the discount and the definition of the promotional context should be separate. The reason for this is simply good separation of concerns. As discussed above, there are several variations possible both for the context definition itself and the discount aspect that depends on it. In an evolution scenario, the shopping company may either change when the discounts are applied without changing the definition of the promotion, or vice-versa. Also, the definition of the promotion may apply to other aspects besides the discount aspect, such as an advertising aspect which adds customized banners for each customer to the shop's web pages. Hence, separating aspects that are dependent on some contexts from the definition of contexts themselves serves well-established engineering principles. In addition to this logical separation between aspects and contexts, contexts should be:

- **stateful**: a context may have state associated to it. Support should be provided for both state internal to the context's implementation (*e.g.*, the time slots when a promotion is active) and publicly accessible state, such as for variation R2 in the example when the rate of discount depends on the promotion. In that case, the discount aspect would define when to actually apply a discount, and would get the specific discount rate from the promotion which defines when this rate is determined. Of course, the state of a context may change dynamically.
- **composable**: a context may be defined from more primitive contexts. For instance, one can define the stock overload context independently of the promotional context; this way, other contexts can be based on stock overload, and the promotional context can combine the time-based definition with the stock overload context.

[1] We also use the interchangeable phrases "the application is in a given context" and "a given context is active".

– **parameterized**: contexts can be defined generically, and parameterized by aspects that are restricted to it; for instance, the stock overload context can be parameterized by the actual threshold that leads to the context being active.

Finally, a context can be related to *control flow properties* (*e.g.*, the web-services-based promotional context), in the broad sense of the term (not only as AspectJ's cflow, but also as past event sequences [11, 13]).

Contextual Restriction. Restricting an aspect to a particular context requires the possibility to refer to a context definition in a pointcut definition. For instance:

```
pointcut amount(): execution(double Item.getPrice()) && inContext(PromotionCtx);
```

Furthermore, since the discount rate may vary on the actual promotion, the associated advice may need to access some external state of the considered context:

```
aspect Discount {
  pointcut amount(double rate): execution(double ShoppingCart.getAmount())
                                  && inContext(PromotionCtx(rate));
  double around(double rate): amount(rate) { return proceed() * (1 - rate); }
}
```

The code above assumes that using rate in PromotionCtx(rate) relies on the fact that a promotional context exposes a rate property (*e.g.*, via a getRate accessor). In addition to context state exposure, it has to be possible to *parameterize* a context when expressing a dependency on it: *e.g.*, Discount could depend on both a time-based PromotionCtx, and the StockOverloadCtx, parameterized by a threshold of, say, 80%[2].

```
  pointcut amount(double rate): execution(double ShoppingCart.getAmount())
            && inContext(PromotionCtx(rate)) && inContext(StockOverloadCtx[.80]);
```

The above inContext pointcut restrictor is semantically equivalent to an if restrictor in AspectJ: restricting an aspect based on whether a certain (context) condition is *currently* verified. However, it should also be possible to restrict an aspect based on whether the application *was* in a certain context previously. There is one generally-useful past context dependency: the context during which an object is created (*a.k.a.*, its *creation context*), *e.g.*, the context during which a shopping cart is created. Using an appropriate language construct createdInCtx, the following implies that Discount applies for any shopping cart that *was created* when PromotionCtx was active, without considering whether the promotion is still active when the customer checks out:

[2] We assume the syntax: (..) for exposing context state, and [..] for context parameterization.

```
pointcut amount(): execution(double ShoppingCart.getAmount())
                   && createdInCtx(PromotionCtx);
```

Finally, it should also be possible to define *domain-* or *application-specific* context restrictors; *e.g.*, to refer to the context during which an item was added to the shopping cart (assuming an appropriate application-specific putInCartInCtx pointcut restrictor):

```
pointcut amount(): execution(double Item.getPrice()) && putInCartInCtx(PromotionCtx);
```

The combination of past context dependencies and stateful contexts implies the need for keeping track of past contexts and their associated state: on the one hand, contexts can be stateful and this state can vary over time; on the other hand, the actual application of the discount can depend on a past promotion context. We refer to this process of keeping track of past contexts and their states as *context snapshotting*, and to the frozen state of a context at a given point in time as a *context snapshot*. A *global* context snapshot is therefore a snapshot of all defined contexts at a given point in time.

For implementation considerations (basically, memory usage), it is obviously impossible to keep the global context snapshots at each and every point in time. It is thus important to be able to snapshot contexts only when really needed. For instance, snapshotting the creation context of an object is needed only if this object is affected by an aspect that is subject to a creation-context dependency. It is also desirable that snapshots be associated to the object they relate to in order to avoid maintaining huge global hashtables, as these would surely become bottlenecks.

Finally, the possibility of defining domain- and application-specific context restrictors implies that the corresponding snapshotting (when to snapshot, where to store the snapshots) be user-definable.

2.3 Summary

The above design analysis of the running example points out a number of requirements for appropriately modeling contexts on the one hand, and for providing aspect language features for context awareness on the other hand, which can be summarized as follows:

- Contexts and context-aware aspects must be separate entities.
- Contexts should possibly be parameterized, stateful, and composable.
- Context state should possibly be bound to pointcut variables.
- It should be possible to express dependencies on past contexts.
- New context-related constructs (possibly domain-specific) should be definable.
- Non-naive implementation of context snapshotting must be supported.

3 An Open Framework for Context-Aware Aspects

We now present an open framework for context-aware aspects that meets the above requirements. The framework is an extension of Reflex [3], a versatile kernel for multi-language AOP in Java [14]. Reflex supports AOP-like dynamic crosscutting, and the framework supports plugins that compile languages such as AspectJ to standard Java programs using Reflex [19]. We do not discuss concrete syntax extensions for supporting context-dependent restrictions here; we focus on the extension of the framework.

3.1 Why a Framework Approach?

A major requirement we have identified is to be able to add new constructs to an aspect language, along with their corresponding semantics. Such an extensible aspect language can be achieved using one of the following alternatives:

- modifying the *interpreter* of the aspect language;
- using a *reflective* aspect language, *i.e.*, an aspect language that has an account of itself embedded within it;
- using an extensible *compiler* for that aspect language.

To the best of our knowledge, there have been no real reflective aspect language proposed to date. As regards extensible compilers, we could implement our proposal using abc, the AspectBench compiler, which is precisely an extensible AspectJ compiler [3]. Our framework-based approach has an advantage in terms of simplicity of the implementation and hence rapidity in prototyping new language features. There is no doubt though that abc would lead to a more efficient implementation as a number of static optimizations could be done: but this is not the purpose of this paper; optimized implementations of context-aware aspects is deliberately left for future research. It has to be noted that working at the framework level is conceptually equivalent to working at the level of the interpreter of the aspect language.

The following section gives a brief overview of Reflex, to give the necessary background for understanding the description of the extended framework that follows. The presentation of the extension for context-dependence is divided in four parts, following the organization depicted in Fig. 1: we first present the context definition framework, followed by the context restriction framework, and we illustrate the use of both with user-defined extensions.

3.2 Background on Reflex

Reflex is an open reflective extension of Java that supports both structural and behavioral modifications of programs. The core concept of Reflex is the *link*; we hereby only need to explain behavioral links: a behavioral *link* invokes messages on a *metaobject* at occurrences of operations specified by a *hookset* [14, 20].

[3] http://reflex.dcc.uchile.cl

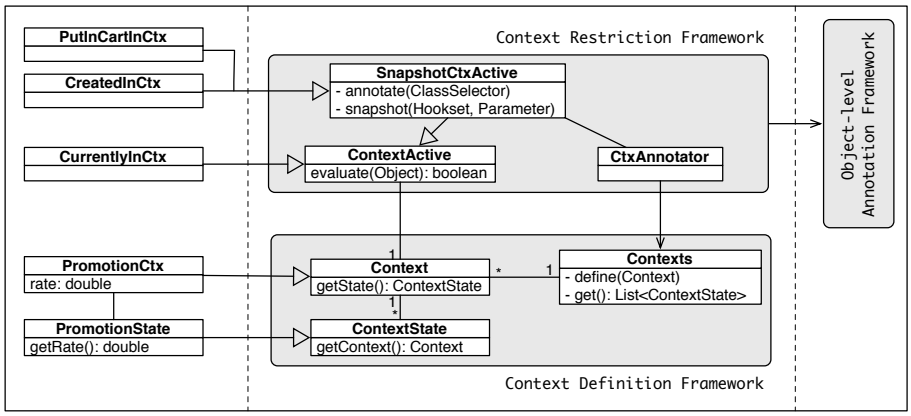

Fig. 1. The Reflex-based frameworks for context-aware aspects: the Context Definition Framework for defining when contexts are active and the Context Restriction Framework for defining context-dependent activation conditions.

A hookset, like an AspectJ pointcut, is a composable entity that specifies a set of operations based on selection conditions. However, it expresses lexical crosscutting only (pointcut shadows). An example hookset and its equivalent in AspectJ is the following:

```
// execution(* WebServiceRequest+.*(..))
 Hookset hs = new Hookset(MsgReceive.class,
              new NameCS("WebServiceRequest", true), new AnyOS()));
```

The selection conditions of a hookset are split into a selection on the type of operation, on the classes in which they occur and on the operations themselves. In the example, the hookset intercepts occurrences of MsgReceive operations, which corresponds to the execution join point type in AspectJ. The NameCS class selector specifies that the class should be WebServiceRequest – or one of its subclasses, as specified by the true parameter, which corresponds to the + in AspectJ. The operation selector AnyOS simply specifies no further conditions on operation occurrences (similar to the use of "*" wildcards).

The exact message sent to the metaobject, as well as how and even if it is really invoked, can be further controlled by setting attributes of the link. Following is an example link definition (amountHS is the hookset corresponding to the amount pointcut):

```
BLink discount = Links.createBLink(amountHS, new Discount());
discount.setCall(Discount.class, "amount");
discount.setControl(Control.BEFORE);       // before advice kind
discount.setScope(Scope.GLOBAL);           // singleton metaobject
discount.addActivation(new Condition());   // adding a dynamic condition
```

The discounting aspect's advice is here modeled as a metaobject, instance of the plain Java class Discount with an amount method, that simply applies the discount. The setCall invocation specifies that the link should send the amount message, with no arguments. The next two invocations illustrate setting the control and scope attributes, which correspond to advice kind and aspect instantiation, respectively.

Important for this paper is that the final invocation specifies an *activation condition*. The activation condition is implemented as an object implementing the interface Active which simply specifies a boolean method evaluate which takes exactly one parameter, the object in which the operation occurred. At runtime, interception occurs only if the activation condition evaluates to true. Hence, an activation condition basically plays the role of pointcut residues in AspectJ.

The messages invoked by links on metaobjects can take parameters. One can specify on a link what arguments to pass by giving a number of Parameter objects. There are several possible objects one can pass, but a number of predefined parameters are available, such as Parameter.THIS and Parameter.RESULT which represent the currently-executing object and the result of the intercepted operation, respectively.

3.3 Context Definition Framework

A simple usage of Reflex for defining context can represent contexts as objects with a boolean method active. Since Reflex supports first-class pointcuts via hooksets and activation conditions, AOP features can be used for defining when a context is active. For instance, consider the definition of the web-service-based promotion context:

```
class PromotionCtx {
  // cflow(execution(* WebServiceRequest+.*(..)))
  CFlow cf = CFlowFactory.get(new Hookset(MsgReceive.class,
                              new NameCS("WebServiceRequest", true), new AnyOS()));
  boolean active() { return cf.in(); }
}
```

The CFlow object exposes the control flow of the hookset given to the CFlowFactory The active method is defined in terms of the in message of this object, which returns true when the application is in the control flow of an operation that matches the hookset.

We also need to define how contexts are snapshot: at any point in time, we may need to snapshot a context, in order to access its state later. One way to design this is to make contexts cloneable, and clone all active contexts when doing a snapshot. But this introduces the issue of managing the depth of context cloning; also, some state of the context is related to its initial parameterization and hence does not need to be cloned.

We rather opted for a design that forces context implementors to explicitly consider the issue of context snapshots. Context definitions should extend the abstract class Context, which instead of having a boolean active method requires

overriding a getState method. This method should return null if the context is not active, otherwise it should return a snapshot of the context as a ContextState object. ContextState is defined as an inner class of Context to ensure the snapshots have a relation to the originating context:

```
abstract class Context {
  Context() { Contexts.define(this); } // define context (explained later)
  abstract ContextState getState(); // null if inactive
  class ContextState {
    Context getContext() { return Context.this; }
}}
```

Suppose that PromotionCtx is characterized by a variable discount rate:

```
class PromotionCtx extends Context {
  CFlow cf = /* as above */
  double rate; /* with setter */
  ContextState getState() {
    if(!cf.in()) return null;
    return new PromotionState(rate);
  }
```

```
class PromotionState
                extends ContextState {
  double rate; // init in constructor
  double getRate() { return rate; }
}}
```

The above implementation ensures that the promotion context can be snapshot correctly: a PromotionState object holds the value of the discount rate at the time the snapshot was requested – although the *current* discount rate may be different.

The framework also includes a global context dictionary, Contexts, which keeps track of all contexts that have been defined (via define(c), as done in the constructor of the abstract Context class) and which can be asked for a global snapshot of all currently active contexts (via get). A global context snapshot is represented as a Snapshot object, which is simply a map of Context-to-ContextState objects. Recall that each context state object contains a reference to the context object that generated it.

3.4 Context Restriction Framework

The context-dependent pointcut restrictors of Section 2.2 can be defined in the Reflex framework as activation conditions on links. Two abstract classes implementing the Active interface are defined as a framework for defining context-dependent activation conditions (Fig. 1). We show here how these are specialized for defining the currentlyInCtx restrictor. The createdInCtx and putInCartInCtx restrictors are presented in the next section. First of all, the CtxActive abstract class is defined as follows:

```
abstract class CtxActive implements Active {
  Context itsContext;
  CtxActive(Context c){ itsContext = c; }  // associate context to condition
  boolean evaluate(Object o){ return getCtxState(o) != null; }
  abstract ContextState getCtxState(Object o);
}
```

In the constructor, the context on which the activation condition relies is stored in an instance variable. The activation condition evaluates to true if the associated context is active, or more precisely, is *determined* to (having) be(en) active by getCtxState. This method is abstract because some context-related conditions rely on a *past* context state, and should therefore be able to retrieve the appropriate context snapshot associated to the currently-executed object. This is explained in more details in the rest of this section.

The CurrentlyInCtx condition restricts a link to be active only when the condition's context is active. So getCtxState returns the value of the context's getState method:

```
class CurrentlyInCtx extends CtxActive {
  ContextState getCtxState(Object o){ return itsContext.getState(); }
}
```

The following is an example of using this activation condition to restrict a link to activate only when the promotion context is active:

```
discount.addActivation(new CurrentlyInCtx(new PromotionCtx()));
```

Past Dependencies. The abstract class SnapshotCtxActive provides the necessary support for defining activation conditions that depend on past context snapshots.

```
abstract class SnapshotCtxActive extends CtxActive {
  ContextState getCtxState(Object o){
    Snapshot snapshot = getSnapshot(o);
    return snapshot.get(itsContext);
} }
```

When queried, such an activation condition extracts the relevant snapshot from the currently-executing object passed as parameter (getSnapshot), and queries the snapshot for the state associated to its context.

There are basically two design options for storing the snapshots of contexts associated with specific objects. One is to rely on a global map of objects to contexts, another is to associate snapshots as extra hidden state in the objects themselves. We opted for the latter option, to avoid a central bottleneck and also since it ensures that the snapshots are removed from memory when the objects they are associated with are garbage collected. We rely on a general object-level annotation framework based on Reflex and its abilities to perform structural changes to classes (not presented here due to space restrictions). To control snapshots, SnapshotCtxActive implements a number of utility methods to interact with this framework, such as:

- void annotate(ClassSelector cs) – enable annotations in classes matched by cs.
- void snapshot(Hookset hs, Parameter p) – upon operation occurrences matched by hs, store snapshot context in the parameter p of the occurrence.
- Snapshot getSnapshot(Object o) – return snapshot stored in o.

3.5 User-Defined Extensions

Having presented the whole framework for context-aware aspects, we illustrate its use by showing how new pointcut restrictors are defined. Two examples are given, one for restricting based on the creation context of a particular object, the other is an application-specific one. Since both refer to past contexts, they are extensions of SnapshotCtxActive.

Creation Context. The creationCtx pointcut restrictor is implemented as the CreatedInCtx class for activation conditions. It can be added as an activation condition to the discount link as follows:

```
CtxActive createdInPromo = new CreatedInCtx(new PromotionCtx(), discount);
discount.addActivation(createdInPromo);
```

Note that the discount link itself is passed as a parameter to the activation condition. This is done so that the link can be introspected, in order to determine which instantiations of which classes the discount aspect actually depends on; the snapshotting of the contexts can thus be limited to instantiations of just those classes. In the constructor of CreationCtx below, the class selector of the link is and used to define snapshotting on just the classes described by the selector:

```
class CreatedInCtx extends SnapshotCtxActive {
  CreatedInCtx(Context c, BLink l){
    super(c);
    ClassSelector cs = l.getClassSelector();
    annotate(cs);
    snapshot(new Hookset(Creation.class, cs, new AnyOS()), Parameter.THIS);
} }
```

The class selector cs is retrieved from the link using its getClassSelector method and is used in a call to snapshot. The hookset passed to snapshot describes all invocations of constructors in classes matching the class selector. Context snapshots are consequently stored in the newly-created object (Parameter.THIS).

In Cart Context. We now illustrate how an application-specific pointcut restrictor is defined, by considering the case of the PutInCartInCtx discussed in Section 2.2. In Reflex, the corresponding activation condition would be used as follows:

```
CtxActive putInCartInPromo = new PutInCartInCtx(new PromotionCtx());
discount.addActivation(putInCartInPromo);
```

The implementation of PutInCartInCtx states that Item objects should be annotated, and that context snapshot annotation occurs upon execution of addItem on a shopping cart; The object to be annotated, the item, is the first parameter (NthParameter(1)) of the call:

```
class PutInCartInCtx extends SnapshotCtxActive {
  PutInCartInCtx(Context c){
    annotate(new NameCS("eshop.Item"));
    snapshot(new Hookset(MsgReceive.class, new NameCS("eshop.ShoppingCart"),
                         new NameOS("addItem")), new NthParameter(1));
}}
```

3.6 State Exposure and Parameterization

Since contexts are standard objects, their parameterization is naturally done by passing parameters at instantiation time, or by sending them configuration messages. For instance, restricting Discount to inContext(StockOverflowCtx[.80]) is implemented as:

```
discount.addActivation(new CurrentlyInCtx(new StockOverflowCtx(.80)));
```

Exposing context state, such as the discount rate in:

```
aspect Discount {
  pointcut amount(double rate): execution(double ShoppingCart.getAmount())
                                  && createdInCtx(PromotionCtx(rate));
  double around(double rate): amount(rate) { return proceed() * (1 - rate); }
}
```

is based on the mechanism provided by Reflex to customize the invocation of a metaobject (recall Section 3.2). The above example would be implemented as follows:

```
Context promotionCtx = new PromotionCtx();
CtxActive inPromo = new CreatedInCtx(promotionCtx, discount);
discount.addActivation(inPromo);
discount.setCall(Discount.class, "amount", inPromo.getCtxParam("rate"));
```

The important line is the last one: the specification of the call to amount (the advice) is changed to include one parameter, bound to the value of the rate state property of the promotional context. getCtxParam is a method of CtxActive[4] that returns a custom Parameter object. To evaluate the value of this parameter at runtime, the activation condition inPromo is queried for the context state corresponding to the currently-executing object, retrieves that context state in the adequate snapshot (in this case, the creation context snapshot), and invokes the getRate method on it.

3.7 More Extensions

In the current work we have only considered context dependencies related to the currently-executing object of an operation occurrence. The two constructs we

[4] The implementation of getCtxParam is not shown as this would lead us too far into details. The interested reader is welcome to ask the authors.

have presented for past context dependencies always look for context snapshots in the this: if the currently-executing object has been created in a given context, or put in the shopping cart in a given context, etc. It would be equally interesting to be able to relate to past contexts associated with other objects, such as the target or any of the arguments of an operation occurrence, or even any object that is accessible from the join point. We could define additional activation conditions that are defined to check for context-dependency related to other objects than the this. Another option fitting in the Reflex framework would be to parameterize context-related activation conditions with a Parameter object to describe on which parameter of the operation occurrence the activation condition acts. In that case, the constructors of the activation condition will also have to perform a slightly more complex introspection of the link's hookset to determine on which classes snapshotting should be performed. We expect to research such extensions in future work.

4 Related Work

We now review related work in the area of contexts, and existing proposals to restrict the scope of aspects. The term *context* can be found in different meanings and definitions in many computer science disciplines, such as human-computer interaction and ubiquitous computing. We hereby only focus on the use of context as an element of a programming language.

Context-Oriented Programming. ContextL [6] is a recent CLOS-based approach for Context-Oriented Programming (COP). While we have presented an extension of an aspect-oriented approach, COP is exploring a paradigmatically new approach. A major difference between both approaches is in the way time is dealt with. In the AOP approach an aspect can be made dependent on whether the application was previously in a certain context, while in COP this is reversed: when the application enters a state that qualifies as being in a certain context, code is redefined so that its future execution takes the "aspect" into account. As the two approaches are paradigmatically different, it is difficult to compare them, though one could state that our proposal has a more declarative nature as the conditions under which the application is in a certain context are encapsulated in context definitions, while in ContextL context has to be more explicitly activated.

Context in other AOP Approaches. The term context has also been used in other AOP approaches: the term is meant to denote information associated with joinpoints, but is typically also limited to information directly associated with joinpoints such as arguments of messages, or the control flow "context" of the joinpoint. We have considered the term context in the more general meaning of all the information about the state of a program when a joinpoint occurs, and have also considered the use of context about past joinpoints, while limiting the amount of information that is kept about past contexts.

Technically, this can be realized in any AOP approach like it is done in the implementation of our framework, *i.e.*, by defining additional pointcuts and advices to capture contexts when needed. However, in our framework these additional pointcuts (or hooksets) are *automatically* installed, they do not have to be written by hand (Sect. 3.5). Also, in our approach context dependencies can be checked in the pointcut, not in the advice, resulting in clearer code.

In AspectJ for example, one can implement a context definition such as the promotion context as an object and the discounting as an aspect. The discounting aspect has one advice for applying the discount, and a second one for snapshotting the promotion context when, *e.g.*, a shopping cart is created. But any aspect depending on the promotion context needs an advice to snapshot the context as needed. An attempt to define a more general promotion context snapshotting aspect runs into the problem that it is not possible to analyze the pointcuts of the aspects that depend on the context. So the reusable definition of the snapshotting necessarily has to snapshot the context at all instance creations, leading to unnecessary overhead. In AspectJ, there is also the problem of defining context activation predicates as reusable named pointcuts because such pointcuts can neither be made dependent on a context using polymorphism (pointcut invocations are not late bound), nor can named pointcuts be parameterized with a context object (named pointcuts only have output arguments).

In more advanced aspect languages it may be possible to achieve an implementation of context-aware aspects that is both more reusable and efficient than an implementation in AspectJ. CaesarJ [2] unifies the concepts of aspects, packages and classes in a single construct, and also similarly to SteamLoom [5, 16] supports dynamic deployment of aspects both globally and thread-locally. These features may be useable to achieve an implementation of an aspect framework for context aware aspects. However, we have so far used the aspect language framework Reflex as our aim is to also provide language support for context-aware aspects.

A number of proposals make it possible for aspects to depend on the past execution history, and to refer to state associated to past events [1, 7, 10, 12]. These approaches make it possible to refer to past context, but only consider context in the sense of join point-related information, as above. Conversely, in this work we consider a more general notion of context, whose state can actually be computed arbitrarily. In other words, we can bind any context state to pointcut variables, which can then be used to parameterize advices. Although the extensions considered in this paper are not impossible to realize in, *e.g.*, EAOP, they have not been explicitly considered. Allowing aspects to refer to the full state of the program was also introduced in the user-extensible logic-based language CARMA using object reifying predicates [15]. This work could also be extended to solve context-dependency problems.

5 Conclusion

Handling context-related behavior as aspects allows for better modularization. In this paper, we have analyzed what it means for aspects to be *context aware* and explored the associated aspect language features. We have exposed an open framework for context-aware aspects, *i.e.*, aspects whose behavior is context dependent. This includes the possibility to restrict aspects to certain contexts, both currently and in the past, as well as to parameterize aspect advices with context-related information. Furthermore, our approach makes it possible to define application- or domain-specific context-related restrictors for aspects.

Future work includes experimenting with context-aware aspects in more elaborate scenarios, and implementing a set of contexts in our framework based on a context-awareness toolkit such as WildCAT [8]. This should provide feedback on our aspect language feature approach to handling context awareness. Also, we have started to investigate the provision of concrete syntax for Reflex, which will enable us to provide context-aware aspects using AspectJ-like syntax.

Acknowledgments. Many thanks to Johan Brichau, Pascal Costanza, Maja D'Hondt, Stéphane Ducasse, Johan Fabry, Oscar Nierstrasz and Roel Wuyts for fruitful discussions on context-oriented programming and context-aware aspects. We gratefully acknowledge the support of Science Foundation Ireland and Lero - the Irish Software Engineering Research Centre, and the Swiss National Science Foundation, project "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006).

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, New York, NY, USA, 2005. ACM Press.

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. *Transactions on Aspect-Oriented Software Development*, 2006. To appear.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[4] M. Baldauf and S. Dustdar. A survey on context-aware systems. Technical Report TUV-1841-2004-24, Technical University of Vienna, 2004.

[5] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[6] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming. In *Proceedings of the ACM Dynamic Languages Symposium*, 2005.

[7] Thomas Cottenier and Tzilla Elrad. Contextual pointcut expressions for dynamic service customization. In *Dynamic Aspect Workshop (DAW'05) as part of AOSD'05*, 2005.

[8] Pierre-Charles David and Thomas Ledoux. WildCAT: a generic framework for context-aware applications. In *Proceeding of MPAC'05, the 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, Grenoble, France, November 2005.

[9] A. K. Dey and G. D. Abowd. Towards a better understanding of context and context-awareness. In *Workshop on the What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, The Hague, The Netherlands, April 2000.

[10] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, March 2004. ACM Press.

[11] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[12] Remi Douence, Olivier Motelet, and Mario Sudholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Heidelberg, and New York, September 2001. Springer-Verlag.

[13] Rémi Douence and Luc Teboul. A pointcut language for control-flow. In Gabor Karsai and Eelco Visser, editors, *Proceedings of the 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *Lecture Notes in Computer Science*, pages 95–114, Vancouver, Canada, October 2004. Springer-Verlag.

[14] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, Lecture Notes in Computer Science, Tallin, Estonia, September 2005. Springer-Verlag.

[15] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference of Aspect-Oriented Software Development*, 2003.

[16] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An execution layer for aspect-oriented programming languages. In *Proceedings VEE 2005*. ACM Press, June 2005.

[17] J. Kephart. A vision of autonomic computing. In *Onward! Track at OOPSLA 2002*, pages 13–36, Seattle, WA, USA, 2002.

[18] P. K. McKinley, S. M. Sadjadi, and B. H Kasten, Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.

[19] Leonardo Rodríguez, Éric Tanter, and Jacques Noyé. Supporting dynamic crosscutting with partial behavioral reflection: A case study. In *SCCC*, pages 48–58, 2004.

[20] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[21] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.