# Context-oriented programming: a software engineering perspective

Guido Salvaneschi[a], Carlo Ghezzi[a], Matteo Pradella[a]

[a] *DEEPSE Group*
*DEI, Politecnico di Milano*
*Piazza L. Da Vinci, 32*
*20133 Milano, Italy*
{*salvaneschi, ghezzi, pradella*}*@elet.polimi.it*

**Abstract**

The implementation of context-aware systems can be supported through the adoption of techniques ad the architectural level such as middlewares or component-oriented architectures. It can also be supported by suitable constructs at the programming language level. Context-oriented programming (COP) is emerging as a novel paradigm for the implementation of this kind of software, in particular in the field of mobile and ubiquitous computing. The COP paradigm tackles the issue of developing context-aware systems at the language-level, introducing ad-hoc language abstractions to manage adaptations modularization and their dynamic activation. In this paper we review the state of the art in the field of COP in the perspective of the benefits that this technique can provide to software engineers in the design and implementation of context-aware applications.

*Keywords:* Context-oriented programming, Context, Context-awareness

## 1. Introduction

Context-awareness is a primary issue in emerging fields such as ubiquitous and mobile computing. In the design of context-aware systems some challenges must be addressed. First, adaptation to the current context is often an aspect that crosscutts the application logic, so it is often orthogonal to the main modularization direction. It is therefore difficult to organize the codebase in a way that does not compromise maintainability and separation of concerns. Furthermore, dynamic adaptation to the context requires that an application modifies its behavior at runtime. While this is not difficult to obtain in principle even with traditional techniques, organizing dynamic behavioral change in a systematic and effective way requires careful engineering.

Over the years, several approaches have been proposed to support the design and development of context-aware software, at different abstraction levels. These approaches mainly encompass software architectures, component-based design, and middleware [1, 2, 3, 4, 5, 6]. Context Oriented Programming (COP) [7] was recently proposed as a complementary approach for supporting dynamic adaptation to context conditions. COP provides language-level abstractions to modularize behavioral adaptation concerns and to dynamically activate them during the program execution.

In this work we review the achievements of context-oriented programming and research advances in the perspective of the benefits they can bring to the software engineering community in general and specifically to the engineering of context-aware systems community. While so far COP has been mainly studied from a programming language point of view, we argue that it can empower software engineers committed to the design of context-aware systems with a very powerful approach and tool. The COP paradigm provides an additional dimension to standard programming techniques to dynamically switch among the behaviors associated with each context, such as bandwidth availability, presence of WiFi or data connection, battery level, or current system workload. In addition, COP provides means to dynamically combine different behaviors when all the associated contexts are active at the same time and properly modularize the code for each behavior.

Our work tries to bridge the gap between the programming languages and the software engineering communities, covering the fundamental aspects which can be of interest for a software architect such as the compilation process, modularization, dynamic activation of adaptations, and consistency of behavioral variations.

```
class Storage{

  Cache cache = ...

  // Other methods

  layer logLayer{
    println("Logging");
    getItem(int key){
      // Send info to the log manager...
      return proceed();
  } }

  layer cacheLayer {
    getItem(int key){
      println("Cache Lookup")
      result = cache.get(key);
      if (result == null){
        result = proceed(key);
        cache.put(key,result);
      }
      return result;
  } }

  getItem(int key){
    println("Disk Lookup");
    Object item;
    // Retrieve item from disk...
    return item;
} }
```

```
Storage s = new Storage();

// No active layers
s.getItem(10);

// cacheLayer active
with(cacheLayer){
  s.getItem(10);
}

// logLayer and cacheLayer active
with(cacheLayer,logLayer){
  s.getItem(10);
}


-- EXECUTION --

> Cache Lookup
> Disk Lookup

> Disk Lookup

> Logging
> Cache Lookup
> Disk Lookup
```

Figure 1: An adaptable Storage Server implemented using COP.

Starting from the pioneering work of Costanza and Hirschfeld [8], COP evolved in a variety of solutions addressing in different ways the problems of behavioral variations modularization and their dynamic activation. The proposed languages share the concept of language-level runtime adaptation to context, but interpret the paradigm in different ways. Therefore, at present COP is constituted by a family of languages specifically developed to support context adaptation, with some widely adopted design solutions and many (sometimes radically) different variants.

In the exposition we adopted the following criteria. We set up our analysis centering it on layer-based [7] COP languages, because they represent the majority of the existing approaches and the most influential efforts in the research community. Single implementations which do not fall into this category and present ad-hoc solutions are also introduced, if they present a feature that is particularly relevant to the discussion.

The paper is organized as follows: In Section 2 we introduce the fundamental concepts of COP. In Section 3 we present the main COP flavors that have been implemented so far. In Section 4 we show an overview of COP existing software and application areas. In Section 5 we analyze the related work and in Section 6 we discuss a roadmap for future research. Section 7 draws the conclusions and outlines future work.

## 2. Fundamental concepts

In this section we present the foundations of the COP paradigm. We adopt a top-down approach, which starts from the abstract notion of context and then focuses on behavioral variations that conceptually enable context-aware adaptation.

The notion of *context* traditionally adopted in COP is open and pragmatic: any computationally accessible information can be considered as context [7]. Such a definition can be surprisingly vague, but practice with context-aware systems confirms its validity. First of all, the definition does not limit the context to the information reaching the system from *outside* (i.e., the *environment*), but it also encompasses information originating *inside* the system boundaries, such as performance monitoring or intrusion detection. Moreover, such a general approach does not prescribe

any restriction to the level of abstraction through which the context is represented inside the system. In fact in a typical context-aware application context information is first obtained by sensors in the form of *numerical* observables. For example, the bandwidth consumption on a network interface can be quantified as 100Mb/sec or the processor is observed to be busy 95% of the time. Often these values are abstracted and combined to obtain some *symbolic* observable. For example, the measured processor usage can be associated with the *heavy load* condition.

A point on which COP approaches differ is whether a unique global context exists or different parts of the system can live in separate contexts. The Ambience programming language [9] adopts a model whereby the whole application shares the same global context. This model reflects the intuitive idea that there is only one real-world context. Conversely, most COP languages [10] do not enforce uniqueness of context and therefore different parts of the application, for example different threads, can live in different contexts and therefore adapt their behavior differently. While from a conceptual point of view a unique context leads to a more elegant and intuitive model, the possibility of exploiting multiple contexts in the same application is more flexible in practice. For example it allows each thread of a server-side software to independently adapt to the specific conditions of the client which is currently in charge of.

A key concept of COP is the *behavioral variation* [7], which is a unit of behavior that can be made effective (partially) modifying the overall behavior of the application. A behavioral variation is enabled by means of a variation *activation*. Runtime context adaptation is achieved in COP by dynamically (i.e. during the execution) activating behavioral variations. When multiple variations are active at the same time, they dynamically *combine* to generate the emerging application behavior. In COP the role of variations is twofold. On the one hand, they allow dynamic activation of a behavioral change; on the other hand, they are the *modularization unit* of such behavioral fragment.

Over the years, this conceptual framework has been interpreted in different ways, originating a variety of different solutions. However the model based on *layers* is by far the most widespread. For this reason, through the paper we generally refer to this model, pinpointing the existence of alternative solutions where needed. Layers [8] are a language abstraction which groups *partial method definitions* implementing behavioral fragments conceptually related to the same aspect of the application context. In Figure 1 we show a simplified implementation of an adaptable storage implemented using COP concepts. This is a running example that we adopt throughout the paper. In the example and in the rest of the paper, where not explicitly said, we adopt a Java-like language, properly augmented with the features required for the explanation. Details not essential for the discussion, such as some type declarations or modifiers, are omitted. By calling the `getItem` method, the storage can be queried for a resource, which is by default searched on disk. The `getItem` method is partially redefined inside the `logLayer` layer and in the `cacheLayer` layer. The `logLayer` layer implements logging facilities and the `cacheLayer` layer adds a caching mechanism, which improves the response time. When `getItem` is called on an instance of the Storage class, if no layer is active the original version is executed, otherwise a partial definition in the active layer is executed.

Several solutions have been proposed in COP for layer activation. They are discussed in detail in Section 3. In the example of Figure 1, the `with` keyword activates the given layers for the scoped block. As a convention, the last activated layer comes first in the execution. If more than one layer is active, the partial definitions are dynamically combined to get the resulting execution. For example if the `logLayer` layer and the `cacheLayer` are both active, logging and caching behaviors are obtained at the same time. Layers combination is achieved through the `proceed` keyword, which is similar to `proceed` in aspect oriented programming or `super` in object-oriented languages. Through `proceed`, the partial definition in the next active layer is executed. If no further partial definition is present in the active layers sequence, the original implementation is called.

In most COP languages layers are first class entities in the sense that they can be assigned to variables, passed as function parameters and returned as values. This is the fundamental way in which different parts of the program can communicate the adaptation to be performed.

In the paper we use the following conventions taken from [11]: *layered method definition* are method definitions for which a *partial method definition* is present. These methods are dispatched according to the context-oriented semantics. Standard object-oriented method definitions are referred to as *plain method definition* and are not affected by the presence of layers.

For convenience, we list the existing COP languages in Table 1. For each language, we describe a distinguishing feature that characterizes it, and we refer to the literature for further study.

| Name | Underlying Language | Distinguishing Features | |
|---|---|---|---|
| ContextJ | Java | First COP source-to-source Java compiler. Implements a reflection API aware of COP abstractions. | [7, 12] |
| JCop | Java | Extends ContextJ. Layer activation can be triggered by jointpoint-like events. Declarative layer activation allows to reduce the scattering of `with` statements. | [13] |
| cj | Java | Prototype of a subset of ContextJ. Runs on an experimental ad-hoc Java virtual machine. | [14] |
| ContextJ* | Java | First ContextJ prototype. Based on a Java library, does not require a special compiler. | [7] |
| ContextLogicAJ | Java | Aspect-based prototype for ContextJ. | [15] |
| JavaCtx | Java | Makes COP easier to integrate with existing tools. COP semantics can be introduced in Java by weaving ad-hoc generated aspects. | [16] |
| EventCJ | Java | A DSL allows to specify transitions between active layers triggered by joinpoint-like events. Layer activation is on per-instance bases. | [17, 18] |
| ContextR | Ruby | Implements a reflection mechanism to query for the active layers. | [19] |
| ContextS | Smalltalk | COP in the Squeak programming environment | [20] |
| ContextLua | Lua | Conceived to support behavioral variations in computer games. | [21] |
| ContextPy | Python | COP in the Python language. | [22] |
| PyContext | Python | Context variables offer specific support for contextual state. Implements implicit layer activation. | [23] |
| ContextL | Common Lisp | First layer-based COP language. Introduces the concepts of layer and dynamically scoped activation. | [8, 24] |
| ContextScheme | Scheme | Similar to ContextL. | [25] |
| ContextJS | Javascript | Open implementation: the programmer can define his own layer activation strategy. | [11] |
| ContextErlang | Erlang | Mixes COP and the actor model through per-agent variation activation triggered by context-related messages. | [26, 27] |
| Ambience | AmOS *(1)* | Context objects implement the behavior for each context. Contextual dispatching through multimethods and context objects as implicit parameters. | [28, 9] |
| Subjective-C | Objective-C | A DLS allows one to define constraints and automatically activate behavioral variations. | [29] |
| Lambic *(2)* | Common Lisp | Based on predicate dispatching. Predicates express contextual conditions. | [30] |

Table 1: Existing COP languages. The bold line divides layer-based languages from non layers-based languages. (1) AmOS is a prototype-based object system built on top of Common Lisp. (2) Lambic [31] is an extension to Common Lisp, proposed to combine generic functions and actors.

## 3. Flavors of Context-Oriented Programming

In this section we present COP in more detail, discussing an overview of the features of the available implementations and the variations to the basic model described so far. We adopt the following approach. First we review the *implementation techniques* adopted for COP languages, which have a significant impact on their usage. For example, library-based implementations are easier to integrate with existing projects while ad-hoc source-to-source compilers can be harder to be accepted in an established development system and can break tool compatibility. Then we consider the *dynamic* aspects of COP, i.e. how behavioral variations are activated. In many cases the `with`-based dynamically scoped activation is not adequate; for example due to the design of the application, it may be necessary to perform the activation on single objects rather than on control flows. We analyze the *static* aspect of COP, which is about the way behavioral variations are modularized in the codebase. This point is essential to keep software maintainable and easy to document. We analyze the existing COP *adaptation directions*. In fact adaptable applications require not only adaptation of computation (i.e. context-aware method dispatching) but also adaptation of state as well as a combination of both (e.g. initialization of state when a change of behavior occurs). Finally we give an overview of how COP deals with the problem of *consistency* between dynamically activated adaptations.

The features discussed in this section are summarized in Table 2.

### *3.1. Implementation Approaches*

COP languages have been implemented as extensions to existing idioms by adding the context-adaptation features. The implementation strategies strongly depend on the underlying language. In general, COP constructs require a modification the existing languages from both a syntactic and a semantic point of view. Syntax modification requires the extension of the language to support the new constructs. Semantic modifications require the encoding of the context-aware behavior beside the standard method dispatching mechanism. To accomplish these tasks two approaches have been followed: the use of libraries, (often based on metaprogramming facilities), and the implementation of source-to-source compilers.

#### *3.1.1. In-language Approaches*

A viable solution for the implementation of COP languages is based on the use of libraries, which make COP features immediately available to the programmer. The main advantage of this approach is that libraries seamlessly integrate with the existing language. This makes the introduction of COP in an established development system easier. In many COP library-based extensions, metaprogramming has a central role. Through metaprogramming, libraries can change the pre-defined semantics of a language, introducing COP behaviors such as layer-aware method dispatching. Metaprogramming [33, 34, 35] is a technique that allows one to programmatically inspect the entities that constitute a program, and possibly modify their behavior. Programming languages support different degrees of metaprogramming. In some cases, the interaction is limited to inspection and use of already existing entities. For example, this is the case of Java in which reflection allows one to analyze classes and invoke the methods obtained from the inspection, but it does not allow to dynamically add a new method to the class nor to perform any runtime modification to the class' structure [36]. In other languages, instead, reflection APIs allow not only inspection, but also structural change. For example in Ruby it is common practice to reopen a class at runtime and add new methods [37].

ContextPy[22], a COP extension to Python, uses *function decorators* to implement layered dispatching. Without entering the details, decorators allow a function to be defined and called instead of the original one. Decorators can be used to intercept a call before it executes in similar way to aspect-oriented programming. In ContextPy this is used to execute the code which dispatches the call according to the layer mechanism. Another noticeable example of metaprogramming-based COP implementation is ContextL, a Common Lisp extension and one of the most mature COP implementations. The contextual features were implemented using the CLOS *metaobject protocol* [38], which gives access to the mechanisms internal to the language runtime, such as the method dispatching algorithm.

Library-based implementations are also used to introduce COP concepts in cases in which it is not required to modify the original language semantics. For example, ContextErlang [26, 27] is implemented as an extension to the Open Telecom Platform (OTP), which is a library and a set of guidelines to implement fault-tolerant distributed and concurrent applications. The key concept of the OTP is the *behavior*, a generic module that implements a recurrent pattern. The standard behaviors in OTP include `gen_fsm` for finite state machines, `gen_event` for event handlers and `supervisor` for processes in charge of monitoring other process executions. When the programmer has to implement

| Name | Implementation | Activation Strategy | Modularization |
|---|---|---|---|
| ContextJ | Source-to-source compiler | DSA | LIC |
| JCop | Source-to-source + Aspect compiler | DSA, Declarative layer composition, Conditional composition | LIC |
| cj | Ad-hoc virtual machine | DSA | LIC, CIL |
| ContextJ* | Java library | DSA | LIC |
| ContextLogicAJ | Preprocessor + Aspect compiler | Indefinite activation | LIC |
| JavaCtx | Library + Aspect compiler | DSA | LIC |
| EventCJ | Source-to-source + Aspect compiler | DSA | LIC |
| ContextR | Library/Meta | DSA | LIC |
| ContextS | Library/Meta | DSA, Indefinite activation | CIL |
| ContextLua | Library/Meta | DSA | CIL |
| ContextPy | Library/Meta | DSA | LIC |
| PyContext | Library/Meta | DSA, Implicit layer activation | CIL |
| ContextL | MOP Library | DSA, Global activation | LIC, CIL |
| ContextScheme | Library/Meta | DSA | CIL |
| ContextJS | Library/Meta | Open implementation *(2)*: DSA, Global activation, Per-instance activation | LIC, CIL |
| ContextErlang | OTP Library | Per-agent activation | Variations are Erlang modules |
| Ambience | AmOS Library | DSA, Global activation | CIL |
| Subjective-C | Preprocessor | Global activation | LIC |
| Lambic | Common Lisp Library *(1)* | – | – |

Table 2: Main features of the existing COP languages. DSA stands for Dynamically Scoped Activation, LIC for Layer-in-class, and CIL for Class-in-layer. (1) In Lambic, since context-adaptation is not obtained through explicit contextual entities (such as layers), the *activation strategy* and the *modularization* features do not apply. (2) ContextJS is an open implementation [32] which allows one to design the activation strategy that bests fits his needs. The strategies in the table are already implemented and ready to use. Part of this table is taken from [10].

e.g., a state machine, she implements the application-specific functionalities, while all the code needed for state transition, fault tolerance and concurrency is provided by the library in the behavior. ContextErlang COP features are implemented in a new behavior, which extends the OTP library, supporting the definition of *context-aware agents*. Context-aware agents are reactive agents [39], which in addition to performing computation upon receiving standard messages modify their functionalities when special context-related messages are received.

In some cases libraries have been used to implement initial COP language prototypes before developing source-to-source compilers. For example ContextJ* falls in this category and is discussed in the next section.

### *3.1.2. Compilers*

Source-to-source compilers have been used to implement several COP extensions. In particular, this solution was adopted for those languages lacking powerful reflective capabilities for which the introduction of the COP constructs was difficult. Source-to-source compilers map the contextual code to standard code in the original language, rearranging the source structure in order to implement the context-adaptation features. The obtained source is then compiled in a traditional way. To the best of our knowledge, ContextJ [40, 12] was the first language to exploit this technique. Interestingly, even in a static language like Java it is possible to simulate COP constructs. For example, in ContextJ*, a prototypal library-based COP Java extension, the `Contextj` class exposes the COP functionalities. The `with` statement is implemented as a statically imported method from that class. The code block evaluated in the scope of the layer activation is simulated using an anonymous class. To give an idea of the impact on readability, we show how the activation of the `caching` and of the `logging` layers of Figure 1 looks like:

```
with (Layers.cacheLayer, Layers.logLayer) .eval(new Block() {
        public void eval() {
          s.getItem(10);
        }
});
```

Of course, the implementation of a source-to-source compiler allows one to get rid of the code burden required to simulate the COP constructs. This significantly increases the readability. The same piece of code in ContextJ would be by far cleaner:

```
with (cacheLayer, logLayer) {
  s.getItem(10);
}
```

Using source-to-source compilers, COP directives can be implemented directly, without the constraint of combining the existing constructs in some eye-pleasing way. Therefore compiler-based implementations can achieve better performance. For example in the transition from ContextJ* to ContextJ a significant speedup was observed [10]. Additionally, as already seen, library-based techniques often employ metaprogramming, which is notoriously inefficient.

A disadvantage of the compiler approach is that compatibility with existing tools supporting the development process is almost certainly broken. The compilation process not only adds new keywords, but also disrupts the original source structure. Therefore tools such as syntax-aware IDEs, code coverage analyzers and performance profilers require to be modified. In source-to-source compilation, additional parameters can be added to method signatures, or the methods themselves can be renamed [12]. The results of the execution of tools on target code usually have no meaning or require to be manually mapped to the structure of the contextual code.

Besides source-to-source compilers, aspect compilers are particularly relevant in COP. Aspect compilers have gained a central role in COP with the recent investigation of activation mechanism other than the dynamically scoped one, such as *declarative layer composition* and *event-driven layer transitions* (Section 3.2). These mechanisms share the idea of automatic layer composition when a certain event occurs. The events are declaratively defined pointcut-like conditions[1], which can be triggered during the execution. JCop [13] and EventCJ [18, 17] support this type of activation and both are implemented as source-to-source compilers using AspectJ as a target language.

JavaCtx [16] is a context-oriented implementation of Java, based on aspects. The goal of JavaCtx is to make the development of COP applications easier and seamlessly integrated with existing tools. JavaCtx receives in input

[1] In aspect-oriented programming, the developer can use predicates called *joinpoints* to specify certain points in the flow of program execution. A set of join points usually combined in a boolean expression is described as a *pointcut* [41].

plain Java, and automatically generates the aspects implementing the contextual semantics. The program is written according to a set of coding conventions to express the contextual constructs. The contextual semantics can be *injected* by weaving the generated aspects through an aspect compiler. With JavaCtx, COP applications are developed using plain and still semantically valid Java code compatible with existing IDEs. The contextual semantics injection is not disruptive, in the sense that it only applies to method dispatching *from outside*, without deeply altering the code structure. This makes tools like a code coverage analyzer or a profiler still usable. Of course, even in JavaCtx the introduction of COP is not completely transparent because the use of aspects has an impact on the development process and on the use of existing tools. For example, a library containing the types used by the aspect weaver is required to execute the application.

### *3.2. Activation mechanism*

Behavioral variation activation as originally proposed in COP is performed through the `with` statement and is dynamically scoped. Over the years researchers have recognized the need for activation mechanisms other than with-triggered dynamic scope. In this section we present the most important solutions.

The choice of which adaptation mechanism to use depends on the design of the application. For example if the adaptation concerns only specific entities that require local modifications, per-object activation can be the best solution. Otherwise, if the adaptation must be performed on all the objects in control flow, dynamically scoped mechanisms allow the activation to be locally controlled and let the behavioral adaptation propagate along the flow of execution. In the sequel, we provide examples and comments that can help one clarify the peculiarities of each solution. However the proper choice depends on the specific application and requires designer expertise.

The freedom of choosing the activation mechanism that best fits the programmer's needs is limited by the fact that not all the solutions are available in each language. It is worth noticing, however, that several languages are based on Java. Thus, many activation mechanisms are available for this mainstream language. A recent research effort [11] considered the option of providing an open implementation the programmer can rely on to develop the activation mechanism which best fits his needs.

#### *3.2.1. Dynamically scoped activation*

Dynamically scoped activation is the solution proposed by most initial COP languages [8, 7]. The `with` statements activate a sequence of layers in the code block. The activation has dynamic extent in the sense that it maintains its effect also in nested calls, and obeys a stack-like discipline. If – in the flow of the execution – another `with` statements is reached, it enables a layer configuration which is composed of all the already active layers, plus the newly activated layer. When the scope expires, the previous configuration is restored. Parallel activation of several layers, such as `with(`*Layer1, Layer2, ...*`)`, has the same meaning of nested single-activation `with` statements, which individually activate each layer. Many COP languages implement also a `without` statement that removes the specified layer.

```
class Storage{

  layer logLayer{
    getItem(int key){
      println("Logging");
      return proceed();
  } }

  getItem(int key){
    println("Disk Lookup");
    ...
} }
```

```
class StorageManager{

  layer logLayer{
    manage(){
      println("Manager");
      proceed();
  } }

  manage(){
    ...
    Storage s = new Storage();
    s.getItem(10);
} }
```

```
StorageManager sm =
  new StorageManager();

sm.manage();
with(logLayer){
  sm.manage();
}

--EXECUTION--
Disk Lookup
Manager
Logging
Disk Lookup
```

Dynamically scoped activation exhibits a remote effect, since the adaptation propagates with the flow of execution. This approach is convenient when all the entities in the control flow are context-dependent. Consider a graphical environment with a complex widget made of several subwidgets. When the main component is asked to draw itself, in turn it asks its subwidgets to do the same. Suppose that each widget can adapt its graphics to the environmental conditions and draw itself in higher contrast when the external light is bright. This situation can be easily managed

by calling the drawing method on the main widget from inside a `with` statement, which activates the `bright` layer. In this way all the widgets are automatically adapted. Note that thanks to dynamic extent, the adaptation is automatically deactivated when the scope expires. This allows the programmer to properly confine adapted regions of the program.

*Declarative layer composition* is a dynamically scoped activation mechanism introduced in [13]. The goal of declarative layer composition is to reduce the spreading of `with` statements that can derive from simple dynamically scoped activation. For example, graphical interface classes usually expose callback methods to capture user actions. If these classes must adapt to the current context, all possible actions must be carried on according to the active layers. This results in several `with` statements scattered across all the callback methods. Declarative layer composition allows one to quantify over control flows, defining joinpoint-like conditions to enable layer activation. This solves the problem of scattered `with` statements. In the following example, layers are declared to be activated in the body of the methods `getItem`, `isItemPresent`, and `getItemSize`. The layers to activate are obtained through the `getActiveLayers` getter method.

```
in(Storage s) &&
(
  on(* Storage.getItem(..)) ||
  on(* Storage.isItemPresent(..)) ||
  on(* Storage.getItemSize(..)) ||
)
{
  with(s.getActiveLayers());
}
```

The `on` keyword identifies an event in the program execution. The predicates that can appear inside the `on` expression are similar to those used in aspect-oriented programming (AOP). When the event is triggered the layer activation in the code block is performed. The `in` keyword binds the current object to a local variable.

*Conditional composition* was introduced to perform variation activation, reacting to asynchronous context-change events [13]. Events are defined as predicates that are evaluated every time a method call, potentially affected by the COP semantics, is executed. If the predicate evaluates to true, a dynamically scoped layer (de)activation is performed.

```
context CacheContext{
  in (Storage s) &&
  when(Storage.isCacheOn()){
// Predicate
  {
    with(cacheLayer);
    // and/or without(...);
  }
}
```

In the example we suppose that the `Storage` class implements an isCacheOn method to inspect if the cache is active. The `when` keyword introduces the dynamically evaluated expression. To enforce a higher level of consistency, after layer activation, the condition is no more evaluated in the dynamic extent of the first layered method execution. Conditional composition is useful when a control flow must be adapted to external events that can happen unexpectedly, such as user interaction. Using the basic dynamically scoped activation would require to continuously check the condition and scattering several `with` statements along the control flow.

### *3.2.2. Indefinite activation*

With indefinite activation, a layer configuration may affect the program behavior starting from a point in the program execution, and its effect extends indefinitely. The active layer configuration is changed when a further activation statement is encountered.

```
class Storage{

  layer logLayer{
    getItem(int key){
      println("Logging");
      return proceed();
  } }

  getItem(int key){
    println("Disk Lookup");
    ...
} }
```

```
class StorageManager{

  manage(){
    ...
    Storage s1 = new Storage();
    activateLayer(logLayer);
    s1.getItem(10);
  } }
```

```
StorageManager sm =
  new StorageManager();
Storage s2 =
  new Storage();

s2.getItem(10);
sm.manage();
s2.getItem(10);

--EXECUTION--
Disk Lookup  // s2

Logging      // s1
Disk Lookup  // s1

Logging      // s2
Disk Lookup  // s2
```

Indefinite activation is a powerful mechanism. It can easily lead the programmer to losing control on which portions of the application are adapted, and which variations are currently enabled. For this reason, it is must be used with care. Not surprisingly some languages employing this technique also provide a way to discipline layer activation. For example, Subjective-C [29], which adopts indefinite activation, also allows one to specify constraints among variations (Section 3.5).

Indefinite activation can affect only the thread performing the activation, or can have influence on all the threads of the application. The latter case, referred to as *global activation* [10], requires particular care. In fact activation driven by one thread can happen asynchronously and unexpectedly in the execution of another thread. In general the adapted thread has no guarantees on when adaptations occurs.

Indefinite activation is probably a good solution when the behavior associated with the adaptation heavily cross-cutts the structure of the application, it is not restricted to certain entities, and there is no need to limit its action. An example of this case is the dynamic activation of logging facilities. Usually logging involves most entities of the application and can be asynchronously activated without the risk of inconsistencies.

### *3.2.3. Implicit activation*

In languages enforcing *implicit activation*, before calling a layered method, all the layers in the application are checked for being active. Each layer exposes an `active` method, which states whether the layer is active or not. Since the runtime support needs to be aware of which layers must be queried, layers must register to it.

```
class Storage{

  layer cacheLayer {
    getItem(int key){
      println("Cache Lookup");
        ...
  } }

  getItem(int key){
    println("Disk Lookup");
    ...
} }
```

```
class cacheLayer{

  isActive(){
    if(MemUse.getVal() > 1000){
      return true;
    }else{
      return false;
    }
} }
```

```
Layers.register(cacheLayer);

Storage s =
  new Storage();

s.getItem(10);
// Memory usage 800 --> 1200
s.getItem(10);

--EXECUTION--
> Disk Lookup

> Cache Lookup
> Disk Lookup
```

Implicit activation is currently available in PyContext [23]. Implicit activation is similar to indefinite activation in that layers influence all the instances without scope restriction. However, in the case of indefinite activation, an explicit set of layers is activated. Instead, with implicit activation all the registered layers are checked at each layered method invocation, to determine those currently active. Indefinite activation is an interesting solution from a modularization standpoint since layer activation can be encapsulated inside the layers themselves. However due to the fact that layers can interact in complex ways, the programmer usually needs to control activation more directly, which limits the

applicability of indefinite activation.

*3.2.4. Per-object activation*

Per-object activation allows layer activation to be controlled on single objects. The activated layers affect only the object selected for the activation, without propagating along the control flow.

```
class Storage{
  layer cacheLayer {
    getItem(int key){
      println("Cache Lookup");
        ...
  } }
  getItem(int key){
    println("Disk Lookup");
    ...
} }
```

```
Storage s =
  new Storage();

s.getItem(10);
s.setWithLayer(cacheLayer);
s.getItem(10);

--EXECUTION--
> Disk Lookup

> Cache Lookup
> Disk Lookup
```

Per-object activation is a suitable solution when behavioral variations limitedly crosscut the application structure. Therefore variation activation must not be repetitively performed on several objects, but rather on a few objects that hold all the key adaptation capabilities of the system. The Pedestrian Navigation System (Section 4.1) belongs to this category, since a single object dynamically switches between WiFi or GPS connection, concentrating most of the adaptation capabilities of the application.

*Event-driven per-object layer transitions* is a variant of per-object activation, in which layer transitions are triggered on specific objects by pointcut-like events [17]. In the following example, the `Activate` event is raised by the execution of the `doSomething` method. The event triggers the activation of the `cacheLayer` layer on the `s` object.

```
class Storage{
  layer cacheLayer {
    getItem(int key){
      println("Cache Lookup");
        ...
  } }
  getItem(int key){
    println("Disk Lookup");
    ...
  doSomething(){ }
} }
```

```
declare event Activate(Storage s)
  :after call(void Storage.doSomething())
      && target(s)
  :sendTo(s);

transition MyEvent
  * activate cacheLayer
```

```
Storage s =
  new Storage();

s.getItem(10);
// Triggers the event
s.doSomething();
s.getItem(10);

--EXECUTION--
> Disk Lookup
> Cache Lookup
> Disk Lookup
```

The `declare event` keywords identify an event expressed in AOP style. `sendTo` identifies the object which is adapted. The `transition` keyword introduces a layer transition triggered by the specified event. Layer transitions allow to define a constraint on behavioral modifications, enforcing consistency across context changes. This aspect is further investigated in Section 3.5.

*3.2.5. Message-driven per-agent activation*

Message-driven activation [26], is performed through context-related messages. Context-related messages enable a behavioral variation on the receiver of the message. The activated variation remains in place indefinitely. The execution triggered by subsequent standard messages is affected by the enabled variation. Of course, this activation mechanism only makes sense in languages supporting the agent concurrency model.

```
agent Storage{
  receive(msg){
    // pattern matching on msg
    [getItem,int key] -> this.getItem(key);
      ...
  }
  getItem(int key){
    println("Disk Lookup");
      ...
  }
  ...
}
```

```
cacheVariation{
  getItem(int key){
    println("Cache Lookup");
      ...
} }
```

```
Agent s =
spawnAgent(Storage);

// sending messages
s -> [getItem,10];
s -> [activate,cacheVariation];
s -> [getItem,10];

--EXECUTION--
> Disk Lookup
> Cache Lookup
> Disk Lookup
```

Per-agent activation is particularly suitable in highly concurrent systems, since the adaptation mechanism is integrated in the concurrency model. Context change is operated through messages which asynchronously intersect the main control flow. So this model allows one to tackle the problem of asynchronous context provisioning in a simple and clean way [27].

### *3.3. Variation Modularization*

Besides dynamic variation activation, COP directly addresses the problem of modularizing context-related adaptations. Since COP support must integrate with the existing language, the modularization technique strongly depends on the underlying idiom.

In the *layer-in-class* model, a layer definition is spread among many modularization units. This model is adopted in languages which already enforce a main modularization direction. The introduction of layers must not break this design. For example, most COP Java implementations (Table 2) adopt this model: since in Java the code is organized along classes, they are preserved as the main modularization direction. Each class declares the partial method definitions belonging to different layers (Figure 1). So partial definitions are modularized together with the layered methods they augment.

In the so-called *layer-in-class* model, layer declarations are outside the lexical scope of the code unit they alter. This model is usually adopted in languages which does not strictly enforce a modularization policy based on language constructs. The layer-in-class approach reminiscent of AspectJ, where aspects are defined outside the classes they augment. As an example of the layer-in-class model we show a fragment of ContextL [8]. In Common Lisp, methods belong to *generic functions* rather than classes. The following code snippet shows the definition of a `log-layer` layer. A generic function `get-item` declares a set of operations with name `get-item` that can be performed on a single object. Method declarations specialize generic functions on certain objects types.

In the example, a layered method definition of `get-item` specialized on `storage` objects is defined (`t` in the `:in-layer` field means "base" layer). A partial definition is then declared in the `log-layer` layer. The `storage` class, potentially affected by the partial method definition, is declared apart.

```
(deflayer log-layer)

(define-layered-function get-item (object))

(define-layered-method get-item :in-layer t ((s storage))
  (format t "Disk Lookup")
  ... )

(define-layered-method get-item :in-layer log-layer :before ((s storage))
  (format t "Logging")
  ... )

(define-layered-class storage ()
  ((cache :initarg :cache)))
```

Which model better fits a language is not always a clear choice. For example, two of the current Python COP extensions make different choices: PyContext adopts the class-in-layer model, while ContextPy adopts the layer-in-class model. This can be considered as a consequence of the fact that Python is an object-oriented and class-based language, but nevertheless exploits *modules* rather than classes as the main modularization unit.

From a software engineering perspective, the main advantage of the class-in-layer approach is that it facilitates software evolution. In fact, adding new behavioral variations only affects those modules defining layers and modifying the already existing modules is not required. On the other hand, with the layer-in-class strategy, variations are declared together with the basic behavior they alter, simplifying the comprehension of the program. Moreover, all the behavioral variations are included in the entity they augment. The resulting code is self-contained and each class encapsulates all its possible behaviors. The layer-in-class solution is more effective in those cases in which it is not possible to recognize a basic behavior and a variation, but the system switches between two or more alternative behaviors. Consider a variant to the storage example, in which the application can pursue the goal of optimizing either the response time or the memory consumption. In this case, the system can be designed with two layers, `optimizePerformance` and `optimizeMemory`. The `getItem` partial definition in the `optimizePerformance` layer stores the items in memory, while the `getItem` partial definition in the `optimizeMemory` layer stores them on disk. So only one layer is active at a time. Of course, switching from a layer to another requires all the items to migrate from memory to disk and vice-versa. The problem of automatically performing initialization operations on layer activations is analyzed in Section 3.4. This example shows that behavioral variations not always are added upon a *basic* behavior and, sometimes, a basic behavior does not exist at all. In such cases, the layer-in-class design allows one to encapsulate the alternative behaviors in classes, improving code modularity and readability.

ContextErlang [26] adopts an hybrid approach. In ContextErlang, each variation is implemented as an autonomous module specifically aimed at modifying the behavior of a certain context-aware agent. Therefore, a certain degree of modularization is maintained, since each context-aware agent is associated with a set of variation modules, all concurring to defining the behavior of the agent. Adding new adaptations is straightforward, since it only requires a new variation module to be provided, without modifying the existing code. Implementing variations as single modules enables an interesting feature of ContextErlang, namely *remote variation transmission*. A context-aware agent running on a remote Erlang node can be unable to adapt properly to an unforeseen situation. In ContextErlang, if another node holds the proper variation, it can send the variation to the remote agent and dynamically activate it. This feature is possible thanks to the Erlang virtual machine, which supports module loading during the program execution.

### *3.4. Adaptation directions*

In a context-aware application, adaptation can be traced back to two aspects: adaptation of *computation*, and adaptation of *state*. The distinction is not crisp in practice, since computation determines access to state and state directly influences computation. However, it is particularly useful for the exposition of the specific COP constructs. Adaptation of computation deals with selection of alternative methods or functions. Adaptation of state deals with the choice among different state values. COP main focus is centered around adaptation of computation (i.e. context-aware method dispatching) rather than state. Nevertheless, some language constructs have been proposed to support state adaptation. In this section, we investigate both adaptation techniques in details, analyzing the variants that have been proposed. Finally, we consider the problem of initialization when a behavioral variation is triggered, an aspect that involves both computation and state.

Layer-driven dispatching has been extensively illustrated in the previous sections. As we observed, this is the solution adopted by most COP languages. However other approaches exists. The Ambience programming language [9] is a based on the Ambient Object System (AmOS), an object system alternative to CLOS built on top of Common Lisp. Ambience, originally developed simultaneously to ContextL, is not layer-based and adopts a global context model. AmOS is based on prototypes rather than classes and delegation links between objects rather than inheritance relations. Ambience leverages multimethods for context-aware dispatching. Multimethods [42], supported traditionally in Common Lisp through generic functions, differ from single dispatching in that all the arguments participate in the method lookup. In AmOS, methods are declared outside objects, similar to standard Common Lisp methods. In Ambience, a context object is passed as an implicit first parameter of each call, therefore methods are dispatched according to the active context. The basic context object delegates to subcontexts, creating a delegation tree. For example an in-depth inspection of the tree can give: the basic context, an *environment* context, an *acoustics* context, and one leaf among the *quiet*, the *normal* and the *noisy* context. When a method is declared to belong to a certain leaf context, it is executed only if the context is currently active, i.e. it is reachable by following the delegation links from the root. Dynamic adaptation is obtained in a straightforward way, by changing the delegation links at runtime.

Predicated generic functions were proposed in [30] as a viable approach to implement context-aware dispatching. Predicated dispatching [43] allows predicates to be expressed in the method definition (e.g. to specify a value for a parameter). When a method is called, the definition whose predicates are fulfilled is actually executed. Traditional dispatching can be considered as a particular case in which the predicates express a condition on the type of the parameters. However, in case of overriding, a problem of ambiguity arises. While types can be linearized to find the most specific type, the problem of computing predicates implications, to find the most specific condition, cannot be decided in the general case. Predicated generic functions alleviate this problem, fostering the declaration of user-defined predicates and the priorities among them in the generic function. In this way, when multiple overriding methods refer to these predicates, there is no ambiguity on which must be executed. Predicates can be easily used to express contextual conditions on the execution of the application, such as the fact that a figure is currently selected, or that the user is moving a shape on the screen. The methods triggered by subsequent user actions, e.g. mouse press or release, are dispatched according to contextual conditions. These features were implemented in Lambic [31], a prototype extension to Common Lisp.

Adaptation of *state* has been explored in ContextL, which allows one to define `:layered-accessors` class slots [2] accessors. Slot accessor methods are only visible when certain layers are active. If access to the slot without the proper layer being active is attempted, an error is raised. Therefore, the same class can declare different state elements associated with different layers. For example in a class representing a graphical shape, the `current-color` slot can be accessible only when the `colored-image` layer is active.

A similar approach can be used in languages which do not support any special contextual state feature. In this case, context-specific state can be simulated by implementing state locations whose access is performed only from computation triggered by that context. For example a private field can be accessed only by a partial method definition, obtaining a sort of layer-specific state.

An interesting combination of COP concepts and state was proposed in PyContext [23] *context variables*. Context variables have context-dependent binding in the sense that the binding of a variable to a value depends on the currently active layer combination. Therefore, context variables' state is preserved across a layer (de)activation i.e., a binding is no more accessible when the active layer combination changes, but it becomes again accessible in case that combination is eventually restored.

The practical implementation of context-aware software raises some problems. When a behavioral change is triggered in the system, it is often needed to perform an initialization of the state before starting the adapted execution. For example, in the storage server of Figure 1, activating the `cacheLayer` layer for the first time may require the `cache` object inside the `Storage` class. Safe initialization has been long recognized as a problem by COP researchers. COP languages were augmented with constructs that allow arbitrary code to be executed when a behavioral variation is activated or deactivated. For example, EventCJ introduced the `activate` and the `deactivate` keywords, which inside a layer definition within a class declare code blocks that must be executed when the layer is activated or deactivated. In the Pedestrian Navigation System (Section 4.1) mobile application, which can use GPS or WiFi connection to get the device location, this mechanism is used to initialize and shutdown the GPS receiver. In a similar way, ContextErlang introduced the `on_activation` and the `on_deactivation` functions inside variations modules.

### *3.5. Behavioral Consistency*

In COP applications, behavioral variations are activated at runtime depending on the contextual conditions. When software grows and becomes complex, the resulting system behavior can evolve in a manner that is not easy to foresee. Moreover, since active variations dynamically combine with one another and with the basic application logic, inconsistencies and conflicts can arise. We refer to this problems as *behavioral consistency*. The COP research community has given several contributions to tackle these issues, devising solutions that can achieve better control over variation activation and enforcing constraints among variations. These approaches encompass reflection, declaration of constraints in the language or through domain specific languages (DSL), and encapsulation of variations management into abstract data types. Despite the importance of the behavioral consistency problem, no general solution is available and most contributions strictly refer to specific language implementations.

[2] In Common Lisp, class *fields* (in Java terminology) or *member variables* (in C++ terminology) are referred as *slots*.

We point out that behavioral consistency is strictly related to variations activation. Activation policies constitute *per se* a form of consistency enforcement among layers. For example, dynamically scoped activation enforces a strict discipline: the layers enabled in the `with` statement are consistently kept active for the whole dynamic scope and automatically removed when the scope expires. It is not surprising that more fine-grained activation approaches, such as *per object* layer activation, required to be regulated by an additional constructs specifying layer transitions. We will also see later that formal verification can be used to ensure activation correctness.

Costanza and Hirshfeld [24] introduced computational reflection for COP languages, which allows one to express constraints on layer activation and to control the fulfillment of constraints during the execution. For example, with layer-aware reflection, it is possible to check if a required layer is already active before activating a layer relying on it. Despite its extreme flexibility, computational reflection is quite complex to master. Further studies proposed to extend ContextL to include *declarative* constraints on layers [44]. This approach makes layer dependency enforcement easier. The violation of a constraint raises an error and the programmer is expected to interactively fulfil the unmet constraint. Thanks to the resumable nature of Common Lisp exceptions (or *conditions*), the execution may subsequently continue. This approach may require human intervention, a fact that limits its applicability in a self-managing autonomic system.

EventCJ allows one to declare *layer transition rules*, which change the active layers of an object when a contextual event is received (Section 3.2). Layer transition rules are constituted by a left-hand side condition on the currently active (or not active) layers, and a right-hand side operation which can use either the `switchTo` or the `activate` operators to specify the layers to activate. The former operator deactivates the layers specified without the `not` modifier in the left-hand side, while the latter keeps them untouched. Transition rules allow layer activation to be controlled with great precision, enforcing constraints between layers. With several events, the behavior given by all the transitions can become complex. The authors of EventCJ propose to model transition rules with finite state automata, and use model checking techniques, namely SPIN [45], to verify safety properties such as constraints on layer activations (e.g.,"layers A and B are never active at the same time").

Subjective-C, a COP extension to Objective-C [29] includes a DSL that can be embedded in the applications to express context dependencies. Subjective-C is based on *contexts* rather than *layers*, resembling Ambience. Similarly to layer-based systems, in Subjective-C methods can be defined to belong to contexts, and dispatching depends on the currently active contexts. In the DSL, four types of relations can be enforced between two contexts A and B. Weak inclusion: the activation of A implies the activation of B but the converse does not necessarily hold. The former context is said to *include* the latter one. For example, *window open* can include *cold*, but *not cold* does not necessarily imply that the window is closed. Strong inclusion: deactivation of B implies the deactivation of A. E.g. *raining* and *cloudy*. Mutual exclusion: A and B cannot be active at the same time. Requirement relation: A requires B to work properly. For example, this code snippet declares the `Offline` and the `Online` contexts and enforces their incompatibility (mutual exclusion is expressed by the `><` symbol).

```
Contexts:
  Online
  Offline
Links:
  Online >< Offline
```

When a context change occurs, the system inspects all the user-defined relations, checking the consistency of the change with the constraints, and possibly triggering new context activation if needed. The DSL solution has probably the advantage of being more intuitive and less verbose than programmatic constraint declaration.

The solution of ContextErlang starts from the observation that when the application is designed, the programmer, in most cases, already knows which use will be done of variations. The ContextChat application [46] is an instant messaging server developed in ContextErlang, where users are implemented as context-aware agents. When users go offline, received messages are stored on the server and delivered later when the addressee connects. An optional backup can be enabled by the user to save messages on a remote server. The user can also provide a custom filter to apply a modification to all its messages, such as changing colors or adding emoticons.

These features are implemented as variations on context-aware agents. For example, the `offline`, `online` and the `backup` variations are provided. In ContextErlang, the programmer does not operate on single variations, but variations are encapsulated in a *context abstract data type* (ADT), which exposes the fundamental operations for their management. The context ADT is organized as a fixed-size stack. Each *slot* of the stack has a name for direct access. Slots can be of three types, depending on the constraints between the variations that can be activated in each of them.

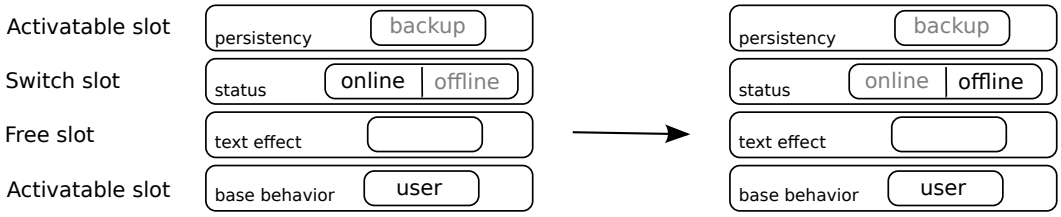


Figure 2: The context ADT in ContextErlang. Switching from the *online* to the *offline* variation.

*Activatable* slots contain a single variation that can be active or not. *Switch* slots contain two or more variations, only one of which can be active at a certain time instant. *Free* slots contain a single variation which is left undefined and can be assigned later. Free slots are required, because in ContextErlang it is possible to send a variation to a remote node, dynamically load it into the Erlang runtime, and activate it on an agent.

In Figure 2 we show the context ADT associated with agents representing a user in the ContextChat server. The `offline` and the `online` variations are in a *switch* slot since they are mutually exclusive and they are never activated together. The `backup` variation is in an *activatable* slot. Optional filters provided by the client are placed in a *free* slot. For symmetry, the base behavior of the agent is placed in a slot like other variations, even though normally it is never deactivated.

## 4. Emerging COP application areas

COP has recently appeared in the landscape of programming languages and the amount of real-world applications currently leveraging the paradigm is rather limited. For this reason, hereafter we consider not only complete implementations, but also proof-of-concept prototypes. We also present running examples discussed in literature when useful.

Most COP applications can be traced back to the fields of ubiquitous and mobile computing. It is interesting to note that COP has proved to be useful not only in this area, which naturally deals with adaptation to the changing environment. In more "traditional" applications, such as text editors, the concept of context-adaptation allows one to easily implement advanced features, achieving cleaner design and better modularity. An overview of the existing applications and a summary of their adaptation capabilities is summarized in Table 3.

### *4.1. PDA devices*

The Pedestrian Navigation System [17] is a mobile application developed using the Android SDK [47]. The application detects the position of the mobile device and displays it on a map. Context awareness is implemented by defining the `GPSNavi` layer and the `WiFiNavi` layer. When the `GPSNavi` layer is active, the current coordinates are obtained using the GPS, and a street map is displayed on the screen. When the user is inside a building, instead, the `WiFiNavi` is active, the position is obtained through a WiFi connection, and the floor plan is displayed by fetching it from a local database.

A scenario which involves smartphones and GPS integration is also described by Gonzales *et al.* as a case study for the Ambience Object System [28, 9]. The CityMaps application stores the map of a city annotated with information for tourists. When a GPS connection is available, instead of using a static map, the application updates the portion of map currently displayed on the screen according to the position of the user. When the GPS module is disabled, the application reverts to the static view. Other phone functionalities are also managed in a context-aware manner. For example a `noisy` context allows the tone of the calls to become louder and the duration of the ring to increase, while the `quiet` context reduces the noise generated by an incoming call.

Gonzales *et al.* developed a set of proof-of-concept applications as a validation of Subjective-C [29]. The first case study simulates a complete home automation system which integrates smartphones as a centralized control for the regulation of temperature, ventilation, lights as well as for control of household appliances. The control application dynamically modifies its behavior and interface according to the current environmental conditions inside the house. A second case study aims at evaluating the use of COP techniques to extend with context-aware features an

| Type | Application | Language | Adaptation | State |
|---|---|---|---|---|
| PDA Devices | Pedestrian Navigation System | EventCJ | The mobile's position is obtained via WiFi or GPS. The GUI displays the street map from googlemaps or from local database accordingly. | C |
| | CityMaps | Ambience | Displays a static map or a dynamically updated map with the user position depending on whether the GPS connection is available. The tone of the calls and the duration of the ring depends on the environmental noise. | R |
| | Home Automation System | Subjective-C | Controls of temperature, ventilation and light depends on environmental conditions. | P |
| | Accelerometer | Subjective-C | Changes bar color and label depending on the device orientation. | P |
| | Data Filtering | Subjective-C | Dynamically switches between different filters applied to the collected data. | P |
| Desktop Software | CJEdit | ContextJ<br>JCop<br>EventCJ | Changes the UI and the reaction to user activity depending whether it is in *commenting* or *programming* mode. | C |
| | LivelyKernel | ContextJS | The testing framework displays additional information when a test is run singularly. Shapes in the workbench can dynamically adapt their appearance and modify their behavior, e.g. because connected to other components. | C |
| | Geuze | Lambic | The effect of user actions (e.g. mouse press) depends on the action context (e.g. drawing shape). | C |
| | Software Transactional Memory | ContextL | The strategy adopted by the STM can be switched dynamically to fit the current needs of the application. | C |
| Server Software | ContextChat | ContextErlang | Chat server. *Online* users forward messages to the client. For *offline* users, messages are kept for future reading. Optional backup and tracing of all messages can be activated for each user. Clients can provide filters for their outgoing messages. | P |
| | Autonomic Storage | ContextErlang | Items can be stored either in memory to reduce response time, or on disk to reduce memory consumption. Optional logging can be dynamically activated. | C |
| | Autonomic Web Application | ContextJ | Each page component can be served in *high bandwidth* or in *low bandwidth* mode to control the network consumption of the web application. | C |



Table 3: COP applications: complete prototypes (C), proof-of-concept implementations (P), running examples (R).

existing application. In this simple case, the original application exploits an accelerometer to display a bar on the screen, keeping the bar always parallel to the ground. The contextual extension simply consist of changing the bar color and label according to the current orientation. Despite the simplicity of the application, this is a first effort in evaluating the integration of COP constructs with existing code. The results show a seamless integration and no apparent performance degradation. A third application is aimed at studying the refactoring with the COP constructs of an existing application, preserving the original behavior and improving the code quality. The application graphs the values gathered by an accelerometer, dynamically switching between the optional filters applied to reduce the noise. COP allows to avoid code duplication in the different operational modes of the application (e.g. low-pass or high-pass) modelling them as different contexts.

### *4.2. Desktop software*

CJEdit is a desktop text editor and programming environment. During the years, it has been used as a case study for the development of new COP languages and it is currently one of the most complete projects in the area. CJEdit was firstly proposed as a case study for ContextJ [40]. The editor allows one to keep documentation and code together in an effective way, enabling rich text formatting for the comments inside a compilation unit. The current context switches between text formatting and code development, depending on the area on which the user is working. The menus, the toolbar and the layout of the application are automatically adapted accordingly. CJEdit was used as a test case for the JCop language [13]. For example the introduction of *declarative layer composition* was motivated by scattered `with` statements in each callback method triggered by a user action. *Event-specific layer activation* were introduced to easily deal with the event-oriented nature of a GUI application. Kamina *et al.* [17] used the CJEdit application as a case study for EventCJ and event-based context transitions. In their implementation the transition between active layer configurations can be stated declaratively and is directly driven by the events generated by the user interaction.

Lincke *et al.* [11] implemented a context-aware unit testing framework in JavaScript. When the whole test suite is executed, the global time spent in the execution is taken and displayed. When the user selects a single test, an adaptation is activated and the time spent in the test is measured and displayed in a separate window. The unit test tool is part of the Lively Kernel [48] which is a platform for web programming written in JavaScript. The Lively Kernel implements the ideas of Morphic [49], an interactive environment that allows one to directly manipulate the graphical elements in the workbench and composing them in more complex objects, with no crisp separation between development and the final graphical result. In [11], the Lively Kernel was used as a case study for experimenting new layer activation mechanisms. Instance-specific layer activation was showed to be useful for the adaptation of each single *Morph* element in the workbench such as rectangles, circles, and text fields.

Geuze [30] is a collaborative graphical editor written in Lambic [31]. The editor allows remote users to collaborate, working on a document at the same time. The authors identify as *context* conditions associated with the editor current usage, such as “painting a shape”, “moving shape”, “drawing selection” or “selecting shape”. The actual method to execute when an action is performed (e.g. “mouse down” or “mouse move”) depends on the context in which the method is chosen. Indeed, dispatching is predicated using the previous conditions. Interestingly, context-aware dispatching was not only used in the development of the graphical user interface, but also in networking, synchronization among users, and replication.

Costanza *et al.* [50] developed a context-oriented software transactional memory using ContextL. Software transactional memories (STM) can adopt different strategies. For example a lock can be required before writing to a memory location, while a copy of the data is kept to restore it in case of a rollback (direct-update). Another solution is that a transaction acquires a copy of the memory and write it back if the transaction commits, avoiding locking at all (deferred-update). The programmer can define some class slots as `transactional`: when they are accessed from inside a code section declared `atomic`, the access is protected. STM strategies can be activated dynamically, depending on which is performing better in a certain application configuration. Activation can be both global or dynamically scoped, allowing each thread to independently adopt its own strategy. The implementation is open, and thanks to the COP modularization facilities, adding a user-defined strategy is straightforward.

### *4.3. Server-side software*

ContextChat [46] is a prototype of an adaptable chat server written in ContextErlang. As briefly introduced in Section 3.5, users are represented inside the server by context-aware agents which act as “avatars” of the user inside

the system. When a client is connected, the associated agent activates the *online* variation, which forwards the received messages to the client. When a client is offline, the *offline* variation is active, storing messages for future reading. Clients can optionally activate a *backup* variation that sends messages to a remote server for persistent storage. A *tracing* variation can be activated by an autonomic manager to collect information on the current message traffic. In a distributed environment, this knowledge can be used to migrate agents which frequently exchange messages to the same node, reducing the overhead of network communications. Finally, thanks to *remote variation transmission* (Section 3.3) a client can provide the associated agent with a variation by implementing a custom filter for outgoing messages, such as adding emoticons or changing the font. The variation is automatically loaded and activated on the remote agent.

The autonomic storage server [46] is prototype of another autonomic application written in ContextErlang. Items can be stored and retrieved using an assigned key. Frequently required items are placed in memory, the others on disk. Each item is implemented as an independent ContextErlang adaptable agent. Adaptability is obtained through the `disk` and `memory` variations, which are dynamically activated by the agent when the item is frequently requested. Additionally, a `log` variation can be activated to trace the application execution.

ContextJ has been used to develop an autonomic web application [51] using servlets inside the Tomcat application server. The application can dynamically modify the amount of network bandwidth it uses. The component of each page (e.g. header, footer, navigation bar) defines the `toString` method, which creates the component representation inside the page. The method is defined both in the *high_band* and in the *low_band* layers. The *high_band* implementation generates a richer page with images and animations, while the *low_band* implementation generates a simpler and less appealing but lighter page. A context manager activates the right layer for each user session. The active layer affects all the components of the page propagating to the subcomponents called in the dynamic scope of the `with` statement. The context manager uses a proportional-integral feedback controller to regulate the fraction of users in each mode. The control is performed according to a defined threshold and the current network bandwidth consumption.

## 5. Related approaches

In this section we review the techniques used to implement adaptive systems, with a special focus on language-level solutions. In this perspective, COP is compared with other similar paradigms such as AOP or other techniques which proved to be effective in the development of context-aware software, like metaprogramming. Finally, we compare COP with feature-oriented programming (FOP) and we trace the concepts of COP back to the first works on layers and multidirectional dispatching. Previous work on comparing COP languages was done in [10]. This paper takes into account recent changes in the COP landscape and assumes a less language-centric perspective.

*Design Patterns.* Design patterns [52] can be used to support the implementation of context-aware software. For example, the *State* pattern consists in a proxy class that exposes a generic interface for a certain behavior. The actual behavior implementation is chosen among concrete classes dynamically linked to the proxy class depending on the context conditions (see e.g. [30]). The State pattern is a "standard" design pattern which can be effectively exploited to implement dynamic adaptation. Riva *et al.* [53] identified extensions to the GOF patterns [52] to specifically tackle the problem of context, and two totally new were proposed. Rossi *et al.* [54] also introduced four new patterns. The proposed patterns solve common issues in the development of context-adaptable software, such as turning existing legacy software into a context-aware application, monitoring the current context of the system, or performing the adaptation according to a set of predefined rules. It is interesting to note that some of these patterns tackle problems which COP directly addresses at the language level. For example, the Typified Context Element pattern in [54] solves the issue of context adaptation through polymorphism and dynamic dispatching, while context-aware dispatching over partial method definitions is natively available in COP. Ramirez and Cheng [55, 56] studied the presence of adaptation-specific design patterns in self-adaptive systems in the context of autonomic computing. They analyzed over thirty adaptation-related research and project implementations, harvesting twelve adaptation-oriented design patterns.

*Aspect-oriented programming.* COP shares some features with AOP [41], such as the availability of special language support to modularize crosscutting concerns. AOP provides a general mechanism for modularization of orthogonal functionalities, while COP is specifically devoted to organizing behavioral variations. More generally,

the focus of AOP is on modularization, while the focus of COP is on runtime activation of variations. This is also confirmed by recent research trends in COP which mainly concentrate on activation mechanisms [13, 17, 11].

Dynamic AOP is about activating aspects at runtime. This allows one to modify software behavior during the execution. Several dynamic AOP frameworks have been implemented, such as CaesarJ [57], Prose [58], JAC [59] and AspectWerkz [60]. These approaches offer general mechanisms for dynamic activation of aspects that can be leveraged to implement *ad-hoc* frameworks to support context-adaptation. On the other hand, COP directly offers the required facilities. Moreover, dynamic features are not widespread among industrial-strength AOP tools. For example control-flow-based activation is supported in the AspectJ language [41], that allows to define `percflow` pointcuts, but this feature is currently lacking in the Spring AOP pointcut model [61].

Dynamic AOP has been introduced in the field of autonomic computing by Greenwood and Blair [62]. Their work highlights the role of Dynamic AOP in adding autonomic behavior to already existing applications, modularizing autonomic features as a crosscutting concern. The TOSKANA [63] toolkit introduces autonomic computing into operating systems using dynamic AOP. Aspect-orientation is used to deploy runtime-activatable aspects into an operating system kernel and modify its behavior while the system is running. Dynamic AOP has been used not only to implement runtime change of behavior, but also to monitor a running application. In this case, sensing operations crosscut the main application logic, and AOP is probably more suitable than COP. For example J-EARS [64] is a framework for autonomic system development and management which allows sensors and effectors to be dynamically added and removed from a working Java application using Dynamic AOP.

*Metaprogramming.* Computational reflection can be exploited to inspect and modify the dynamic behavior of an application. This approach was studied by Dowling *et al.* [65]. They compared different language-level techniques to support software dynamic adaptation: reflection, dynamic link libraries (DLL), and design patterns. They concluded that, at the cost of a non-negligible performance overhead, computational reflection offers significant advantages in separating functional and adaptation code. This study, however, dates back to before the wide spreading of AOP. With the gain in popularity of AOP, the role of metaprogramming in managing separation of concerns became less appealing. AOP is specifically targeted to modularization of crosscutting concerns, but in the non-dynamic version it lacks features supporting runtime modification. It is interesting to observe that AOP was used in conjunction with metaprogramming to fill this gap. For example Sadjadi *et al.* [66] used a mix of AOP and computational reflection to implement TRAP/J, a toolkit that allows adaptive behavior to be added to an existing Java application, without modifying the original source code. Thanks to AOP, the original classes are replaced by wrappers at instantiation time; then, using a meta object protocol, the application behavior is changed at runtime by dynamically redirecting methods calls to delegated objects. TRAP/C++ [67] is a C++ implementation of the same concepts that uses generative programming to overcame the lack of reflective capabilities of the C++ language.

These works show that computational reflection is used to support runtime change, while AOP allows separation of adaptation concerns. COP addresses the problem of variation modularization *and* dynamic behavioral change in a single paradigm. The role of metaprogramming and AOP in self-adaptive systems is investigated by McKinley *et al.* [68], who analyzed in detail the techniques applied to adaptive software composition, identifying three key enabling technologies: separation of concerns, computational reflection, and component-based design.

*Other techniques similar to COP.* Feature-oriented programming (FOP) is about synthesizing programs in software product lines [69] through step-wise refinements from single units of functionality conventionally called *features*. The AHEAD toolsuite [70] is a Java-based implementation of FOP, in which separate files are used to express class refinements. COP and FOP share the idea of supporting variations of the original program with language-level techniques. Since the behavior of a product variant is known in advance, FOP features are normally selected and combined at compile time. COP variations, instead, due to the volatile nature of the context, are activated and combined dynamically. Dynamic Software Product Lines (DSPL) [71] are a recent solution for adaptive systems in which available features are switched at run time [72, 73]. Research in this direction is mostly the architectural level, while COP specifically focus on language-level abstractions.

In object-oriented programming the binding of a method call with the actual implementation depends on two elements: the *message* (i.e. the method name and the actual parameters) and the receiving object. This overcomes the single-dimension dispatch of procedural languages where the function to execute is selected only on the basis of the name and the parameters of the call. Subjective dispatch [74] adds a dimension to the receiver-based dispatching

of object-oriented languages, taking into account also the sender. COP can be considered a solution which enables dispatching along the context dimension.

*Self-adaptive Sofware.* Software engineering research in the area of self-adaptive software mostly addressed the issues of dynamic adaptation from an architectural viewpoint. Research in this direction have investigate architectural styles, in order to find the approach that better supports self-adaptation, and solutions for the problem of runtime change. For example, [75] shows how the C2 architectural style enables dynamic evolution. In C2, adaptation is achieved through addition or removal of components and can be carried out without suspending the computation. The Rainbow framework [76] offers facilities for distributed component monitoring, deployment of sensors and system reconfiguration based on strategies. It provides a reusable infrastructure to support architecture-based self-adaptation, offering a general approach that can be then tailored to a specific class of systems.

Middlewares have been investigated to implement self-adaptation. Sadjadi and McKinley [77] proposed ACT (Adaptive CORBA Template), which allows the weaving of adaptive code into CORBA applications, modifying their behavior at runtime. The injected code operates on messages and exceptions which pass through the CORBA infrastructure. The Madam middleware [78] is able to detect context changes, make decisions about the adaptation to perform and finally implement the architectural changes. Interestingly, instead of *condition-actions rules* such as those used by Rainbow, Madam adopts *utility functions* which establish general objectives that the middleware must fulfill. Reasoning mechanisms are provided to find the best solution to implement the user-defined policies.

## 6. Discussion and Research Roadmap

COP is a recent technique compared to other more established paradigms such as AOP, which is now implemented in many enterprise frameworks. So far, COP research has been mostly carried out by the programming languages community. This established a proper foundation for the emerging field, and helped to identify the peculiarities of COP with respect to existing similar solutions. Several examples were reported that show success of COP in tackling certain programming problems, where traditional techniques fail or are less effective to cope with. However, the portfolio of applications actually implemented with COP is still limited. We believe that one of the important roles of software engineering research is to provide all the conceptual and practical tools that support the development of a software artifact leveraging language abstractions. In this section, we trace a possible research roadmap for COP, analyzing the issues that most evidently require (further) investigation.

*Specification.* The area of specification in relation to COP is still unexplored, and the definition of a precise description of the behavior implemented by each variation is an open problem. This is particularly significant in the context of self-adaptive software. In fact adaptation is usually triggered automatically by some reasoning engine, which is in charge of pursuing high-level goals. Therefore, a formal specification of the behavioral change introduced by each variation is required to effectively plan the activation in a fully automated manner.

A first effort was done in the specification of constraints among layers. Costanza and D'Hondt [44] used Feature Description Language [79] to express layer constraints. An open problem is how to solve possible conflicts. In [44], a violation raises an error which is prompted to the programmer, but in a fully automated environment this is clearly unacceptable. Gonzales *et al.* [29] (see also Section 3.5) used a DSL for expressing layer dependencies and automatically fulfill constraints violations.

*Modelling.* Another emerging issue is the representation of COP abstractions using modelling languages. Costanza and D'Hondt [44] borrowed feature diagrams [80] from feature-oriented programming to model COP layers. Lincke *et al.* [11] represent layers as UML containers marked with the `<<Layer>>` stereotype. In their approach, classes are replicated in multiple layers with the `<<partial class>>` stereotype, to model partial method definitions. `<<partial class>>` definitions are associated with their definition in the `Base` layer through the `adapts` relation. This notation is effective in practice and it was adopted in the work to analyze the structure of a running example. However, a systematic effort to propose a shared notation for the representation of COP abstractions in UML is still needed, either by extending existing notations or using the available constructs.

*Verification and Testing.* Dynamically adaptable software is hard to design, since runtime adaptations can trigger unpredictable behavior. Verification can play an important role in assuring that, even in a highly dynamic environment, the desired properties hold. As already mentioned, Kamina *et al.* [17] verified properties expressing constraints on layer activations using the SPIN model checker. This approach is a first attempt to verify a system implemented in a COP language, taking into account context-related abstractions. However in [17] the specification of the system in Promela (SPIN's language) is written manually and no support is given to automatically translate the DSL for layer transitions to Promela code. Another research direction is the verification to those layer activation mechanisms for which there is no direct mapping to FSMs, such as dynamically scoped activation.

To the best of our knowledge, the area of testing for COP applications is completely unexplored. Traditional testing techniques can be applied to COP programs, but they do not consider contextual constructs, which can enrich the semantics of the analysis result. For example, a first effort in this direction could be the definition of COP abstractions-aware coverage criteria, such as *layer coverage* which allow to analyze how a test suite can stress each layer of the application. Another interesting point is to define testing techniques that are able cover layer combinations, to get some confidence that a certain combination does not lead to runtime errors.

## 7. Conclusion

In this work we presented an overview of the COP techniques in the perspective of software engineering of context-aware systems. We strongly believe that the community of context-adaptive systems research can benefit from the recent advances in COP programming languages.

So far, our main effort was committed to the design and the implementation of ContextErlang. Our plans for the future are threefold. First, we will attempt to make the adoption of COP languages easier. Our work on JavaCtx goes in this direction. We intend to develop and extend JavaCtx, by implementing some advanced functionalities, such as context-specific reflection.

Second, we intend to continue investigating the use of COP in the development of context-aware and dynamically adaptable applications, in the wide scope of the SMSCom (*Self Managing Situated Computing*) project we are engaged in, with the aim of evaluating the COP paradigm in real-world middle-sized projects. The term *situational* indicates that software behaves according to the evolving situation in which it operates, a concept that fully correspond to context in COP. Developing and running situational software imposes a paradigmatic shift from a conventional development to new scenario in which bits of applications are composed in possibly unpredictable ways. At runtime COP appears to be a natural approach for developing this kind of applications.

Third, we intend to systematically analyze the support that traditional non-COP languages can offer for dynamic adaptation. Our work [81] is a starting point in this investigation. In that work we setup an evaluation framework to properly study various languages with respect to dynamic adaptation. The framework considers several dimensions such as the abstractions that allow to express adaptations, how adaptations are combined, and how they are activated. In the future, we will further extend and validate this conceptual framework also by increasing the set of languages and features that our analysis framework takes into account.

*Acknowledgments*

This research has been funded by the European Community's IDEAS- ERC Programme, Project 227977 (SMSCom).

## References

[1] A. T. S. Chan, S.-N. Chuang, MobiPADS: A reflective middleware for context-aware mobile computing, IEEE Trans. Softw. Eng. 29 (2003) 1072–1085.

[2] P. Bellavista, A. Corradi, R. Montanari, C. Stefanelli, Context-aware middleware for resource management in the wireless internet, IEEE Transactions on Software Engineering 29 (2003) 1086–1099.

[3] T. Gu, H. K. Pung, D. Q. Zhang, A service-oriented middleware for building context-aware services, Journal of Network and Computer Applications 28 (2005) 1–18.

[4] T. Gu, H. Pung, D. Zhang, Toward an OSGi-based infrastructure for context-aware applications, Pervasive Computing, IEEE 3 (2004) 66 – 74.

[5] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, E. Stav, A comprehensive solution for application-level adaptation, Softw. Pract. Exper. 39 (2009) 385–422.
[6] L. Capra, W. Emmerich, C. Mascolo, CARISMA: Context-aware reflective middleware system for mobile applications, IEEE Transactions on Software Engineering 29 (2003) 929–945.
[7] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology 7 (2008).
[8] P. Costanza, R. Hirschfeld, Language constructs for context-oriented programming: an overview of ContextL, in: Proceedings of the 2005 symposium on Dynamic languages, DLS '05, ACM, New York, NY, USA, 2005, pp. 1–10.
[9] S. González, K. Mens, A. Cádiz, Context-oriented programming with the Ambient object system, j-jucs 14 (2008) 3307–3332.
[10] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, M. Perscheid, A comparison of context-oriented programming languages, in: COP '09: International Workshop on Context-Oriented Programming, ACM, New York, NY, USA, 2009, pp. 1–6.
[11] J. Lincke, M. Appeltauer, B. Steinert, R. Hirschfeld, An open implementation for context-oriented layer composition in ContextJS, Sci. Comput. Program. 76 (2011) 1194–1209.
[12] M. Appeltauer, R. Hirschfeld, M. Haupt, H. Masuhara, ContextJ: Context-oriented programming with Java, Information and Media Technologies 6 (2011) 399–419.
[13] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, K. Kawauchi, Event-specific software composition in context-oriented programming, in: B. Baudry, E. Wohlstadter (Eds.), Software Composition, volume 6144 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2010, pp. 50–65. 10.1007/978-3-642-14046-4.
[14] H. Schippers, M. Haupt, R. Hirschfeld, An implementation substrate for languages composing modularized crosscutting concerns, in: Proceedings of the 2009 ACM symposium on Applied Computing, SAC '09, ACM, New York, NY, USA, 2009, pp. 1944–1951.
[15] M. Appeltauer, R. Hirschfeld, T. Rho, Dedicated programming support for context-aware ubiquitous applications, in: UBICOMM '08: Proceedings of the 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, IEEE Computer Society, Washington, DC, USA, 2008, pp. 38–43.
[16] G. Salvaneschi, C. Ghezzi, M. Pradella, JavaCtx: Seamless toolchain integration for context-oriented programming, in: Proceedings of the 3rd International Workshop on Context-Oriented Programming, COP '11.
[17] T. Kamina, T. Aotani, H. Masuhara, EventCJ: a context-oriented programming language with declarative event-based context transition, in: Proceedings of the tenth international conference on Aspect-oriented software development, AOSD '11, ACM, New York, NY, USA, 2011, pp. 253–264.
[18] T. Kamina, T. Aotani, H. Masuhara, Designing event-based context transition in context-oriented programming, in: Proceedings of the 2nd International Workshop on Context-Oriented Programming, COP '10, ACM, New York, NY, USA, 2010, pp. 2:1–2:6.
[19] G. Schmidt., ContextR and ContextWiki., Master's thesis, Potsdam, 2008.
[20] R. Hirschfeld, P. Costanza, M. Haupt, An introduction to context-oriented programming with ContextS, in: R. Lämmel, J. Visser, J. Saraiva (Eds.), GTTSE, volume 5235 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 396–407.
[21] B. H. Wasty, A. Semmo, M. Appeltauer, B. Steinert, R. Hirschfeld, ContextLua: dynamic behavioral variations in computer games, in: Proceedings of the 2nd International Workshop on Context-Oriented Programming, COP '10, ACM, New York, NY, USA, 2010, pp. 5:1–5:6.
[22] C. Schubert., ContextPy and PyDCL - Dynamic Contract Layers for Python., Master's thesis, Potsdam, 2008.
[23] M. von Löwis, M. Denker, O. Nierstrasz, Context-oriented programming: Beyond layers, in: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), ACM Digital Library, 2007, pp. 143–156.
[24] P. Costanza, R. Hirschfeld, Reflective layer activation in ContextL, in: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, ACM, New York, NY, USA, 2007, pp. 1280–1285.
[25] ContextScheme website, `http://p-cos.net/context-scheme.html`, 2011.
[26] C. Ghezzi, M. Pradella, G. Salvaneschi, Programming language support to context-aware adaptation - a case-study with Erlang, SEAMS: Software Engineering for Adaptive and Self-Managing Systems, International Workshop, ICSE 2010 (2010).
[27] C. Ghezzi, M. Pradella, G. Salvaneschi, Context oriented programming in highly concurrent systems, in: Proceedings of the 2nd International Workshop on Context-Oriented Programming, COP '10, ACM, New York, NY, USA, 2010, pp. 1:1–1:3.
[28] S. González, K. Mens, P. Heymans, Highly dynamic behaviour adaptability through prototypes with subjective multimethods, in: Proceedings of the 2007 symposium on Dynamic languages, DLS '07, ACM, New York, NY, USA, 2007, pp. 77–88.
[29] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, J. Goffaux, Subjective-C: Bringing context to mobile platform programming, in: Proceedings of the International Conference on Software Language Engineering, 2010, Lecture Notes in Computer Science, Springer-Verlag, Eindhoven, The Netherlands.
[30] J. Vallejos, S. González, P. Costanza, W. De Meuter, T. D'Hondt, K. Mens, Predicated generic functions: enabling context-dependent method dispatch, in: Proceedings of the 9th international conference on software composition, SC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 66–81.
[31] J. Vallejos, P. Costanza, T. Van Cutsem, W. De Meuter, Reconciling generic functions with actors, in: ACM SIGPLAN International Lisp Conference 2009, Cambridge, Massachusetts.
[32] G. Kiczales, Beyond the black box: open implementation, Software, IEEE 13 (1996) 8, 10 –11.
[33] B. C. Smith, Procedural reflection in programming languages, Thesis (Ph.D.)–Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1982.
[34] P. Maes, Concepts and experiments in computational reflection, in: Conference proceedings on Object-oriented programming systems, languages and applications, OOPSLA '87, ACM, New York, NY, USA, 1987, pp. 147–155.
[35] J. Ferber, Computational reflection in class based object-oriented languages, SIGPLAN Not. 24 (1989) 317–326.
[36] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Third Edition, Addison-Wesley Longman, Amsterdam, 3 edition, 2005.
[37] P. Perrotta, Metaprogramming Ruby, Pragmatic Bookshelf, 1st edition, 2010.
[38] G. Kiczales, J. D. Rivieres, The Art of the Metaobject Protocol, MIT Press, Cambridge, MA, USA, 1991.
[39] C. Hewitt, P. Bishop, R. Steiger, A universal modular actor formalism for artificial intelligence, in: IJCAI'73: Proceedings of the 3rd

international joint conference on Artificial intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, pp. 235–245.
[40] M. Appeltauer, R. Hirschfeld, H. Masuhara, Improving the development of context-dependent Java applications with ContextJ, in: International Workshop on Context-Oriented Programming, COP '09, ACM, New York, NY, USA, 2009, pp. 5:1–5:5.
[41] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: J. Knudsen (Ed.), ECOOP 2001 – Object-Oriented Programming, volume 2072 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2001, pp. 327–354. 10.1007/3-540-45337-718.
[42] L. Salzman, J. Aldrich, Prototypes with multiple dispatch: An expressive and dynamic object model, in: A. P. Black (Ed.), ECOOP 2005 - Object-Oriented Programming, volume 3586 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2005, pp. 734–734. 10.1007/11531142_14.
[43] M. D. Ernst, C. S. Kaplan, C. Chambers, Predicate dispatching: A unified theory of dispatch, in: ECOOP '98, the 12th European Conference on Object-Oriented Programming, Brussels, Belgium, pp. 186–211.
[44] P. Costanza, T. D'Hondt, Feature descriptions for context-oriented programming, in: Software Product Lines, 12th International Conference (SPLC), 2008, pp. 9–14.
[45] G. Holzmann, Spin model checker, the: primer and reference manual, Addison-Wesley Professional, first edition, 2003.
[46] G. Salvaneschi, C. Ghezzi, M. Pradella, ContextErlang: Introducing Context-oriented Programming in the Actor Model, submitted for publication, 2011.
[47] Reference website for the Android SDK, 2011. Http://developer.android.com/sdk/.
[48] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, T. Mikkonen, Self-sustaining systems, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 31–50.
[49] J. H. Maloney, R. B. Smith, Directness and liveness in the morphic user interface construction environment, in: Proceedings of the 8th annual ACM symposium on User interface and software technology, UIST '95, ACM, New York, NY, USA, 1995, pp. 21–28.
[50] P. Costanza, C. Herzeel, T. D'Hondt, Context-oriented software transactional memory in Common Lisp, in: Proceedings of the 5th symposium on Dynamic languages, DLS '09, ACM, New York, NY, USA, 2009, pp. 59–68.
[51] G. Salvaneschi, C. Ghezzi, M. Pradella, Context-oriented programming: A programming paradigm for autonomic systems, CoRR abs/1105.0069 (2011).
[52] Gamma, Helm, Johnson, Vlissides, Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley, Massachusetts, 2000.
[53] O. Riva, C. D. Flora, S. Russo, K. Raatikainen, Unearthing design patterns to support context-awareness, Pervasive Computing and Communications Workshops, IEEE International Conference on (2006) 383–387.
[54] G. Rossi, S. Gordillo, F. Lyardet, Design patterns for context-aware adaptation, in: Proceedings of the 2005 Symposium on Applications and the Internet Workshops (SAINT-W'05), pp. 170–173.
[55] A. J. Ramirez, B. H. Cheng, Design patterns for developing dynamically adaptive systems (poster summary), in: Proceedings of the 6th IEEE International Conference on Autonomic Computing and Communications, Barcelona, Spain.
[56] A. J. Ramirez, B. H. C. Cheng, Design patterns for developing dynamically adaptive systems, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '10, ACM, New York, NY, USA, 2010, pp. 49–58.
[57] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, An overview of CaesarJ, in: A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development I, volume 3880 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2006, pp. 135–173.
[58] A. Popovici, G. Alonso, T. Gross, Just-in-time aspects: efficient dynamic weaving for Java, in: Proceedings of the 2nd international conference on Aspect-oriented software development, AOSD '03, ACM, New York, NY, USA, 2003, pp. 100–109.
[59] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, JAC: A flexible solution for aspect-oriented programming in Java, in: A. Yonezawa, S. Matsuoka (Eds.), Reflection, volume 2192 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 1–24.
[60] J. Bon, Aspectwerkz - dynamic AOP for Java, in: Proceedings of the 3rd international conference on Aspect-oriented software development, AOSD'04.
[61] Reference website for the Spring framework, 2011. Http://www.springsource.org/.
[62] P. Greenwood, L. Blair, Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System, Technical Report, Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS, 2003.
[63] M. Engel, B. Freisleben, Supporting autonomic computing functionality via dynamic operating system kernel aspects, in: Proceedings of the 4th international conference on Aspect-oriented software development, AOSD '05, ACM, New York, NY, USA, 2005, pp. 51–62.
[64] P. Bachara, K. Blachnicki, K. Zielinski, Framework for application management with dynamic aspects J-EARS case study, Inf. Softw. Technol. 52 (2010) 67–78.
[65] J. Dowling, T. Schäfer, V. Cahill, P. Haraszti, B. Redmond, Using reflection to support dynamic adaptation of system software: A case study driven evaluation, in: Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999, Springer-Verlag, London, UK, 2000, pp. 169–188.
[66] S. M. Sadjadi, P. K. Mckinley, B. H. C. Cheng, R. E. K. Stirewalt, TRAP/J: Transparent generation of adaptable Java programs, in: Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04), Agia.
[67] S. D. Fleming, B. H. C. Cheng, R. E. K. Stirewalt, P. K. McKinley, An approach to implementing dynamic adaptation in C++, in: Proceedings of the 2005 workshop on Design and evolution of autonomic application software, DEAS '05, ACM, New York, NY, USA, 2005, pp. 1–7.
[68] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, B. H. C. Cheng, Composing adaptive software, Computer 37 (2004) 56–64.
[69] D. Batory, J. N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 187–197.
[70] D. Batory, Feature-oriented programming and the ahead tool suite, in: Proceedings of the 26th International Conference on Software Engineering, ICSE '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 702–703.
[71] C. Cetina, P. Giner, J. Fons, V. Pelechano, Designing and prototyping dynamic software product lines: techniques and guidelines, in: Proceedings of the 14th international conference on Software product lines: going beyond, SPLC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 331–345.
[72] N. Bencomo, P. Sawyer, G. S. Blair, P. Grace, Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems., in: S. Thiel, K. Pohl (Eds.), SPLC, International Software Product Line Conference, Lero Int.

Science Centre, University of Limerick, Ireland, 2008, pp. 23–32.
[73] S. Hallsteinsen, E. Stav, A. Solberg, J. Floch, Using product line techniques to build adaptive systems, in: Software Product Line Conference, 2006 10th International, pp. 10 pp. –150.
[74] R. B. Smith, D. Ungar, A simple and unifying approach to subjective objects, TAPOS 2 (1996) 161–178.
[75] P. Oreizy, N. Medvidovic, R. N. Taylor, Architecture-based runtime software evolution, in: ICSE '98: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, Washington, DC, USA, 1998, pp. 177–186.
[76] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self-adaptation with reusable infrastructure, Computer 37 (2004) 46–54.
[77] S. M. Sadjadi, P. K. McKinley, ACT: an adaptive CORBA template to support unanticipated adaptation, in: Distributed Computing Systems, 2004. Proceedings. 24th International Conference on, pp. 74 – 83.
[78] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, E. Gjorven, Using architecture models for runtime adaptability, Software, IEEE 23 (2006) 62 – 70.
[79] A. van Deursen, P. Klint, Domain-specific language design requires feature descriptions, Journal of Computing and Information Technology 10 (2001) 2002.
[80] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, 1990.
[81] C. Ghezzi, M. Pradella, G. Salvaneschi, An evaluation of the adaptation capabilities in programming languages, in: Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems, SEAMS '11, ACM, New York, NY, USA, 2011, pp. 50–59.