Information and Software Technology 156 (2023) 107132

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

An expressive and modular layer activation mechanism for Context-Oriented Programming

Paul Leger [a,*], Nicolás Cardozo [b], Hidehiko Masuhara [c]

[a] *Escuela de Ingeniría, Universidad Católica del Norte, Coquimbo, Chile*
[b] *Systems and Computing Engineering Department, Universidad de los Andes, Colombia*
[c] *Tokyo Institute of Technology, Japan*

ARTICLE INFO

Keywords:
Context-Oriented Programming
Activation mechanisms
EMA

ABSTRACT

Context. There is a trend in the software industry towards building systems that dynamically adapt their behavior in response to their surrounding environment, given the proliferation of various technological devices, such as notebooks, smartphones, and wearables, capable of capturing their execution context. Context-oriented Programming (COP) allows developers to use layer abstractions to adapt software behavior to the context. A layer is associated with a context and can be dynamically activated in direct response to gathered information from its surrounding execution environment. However, most existing layer activation mechanisms have been tailored specifically to address a particular concern; implying that developers need to tweak layer definitions in contortive ways or create new specialized activation mechanisms altogether if their specific needs are not supported.
Objective. Complementing ideas to expressively declare activation mechanism models with interfaces that define conditionals of activation mechanisms modularly, this paper proposes an Expressive and Modular Activation mechanism, named EMA.
Method. To propose EMA, we analyze existing activation mechanisms in COP regarding activation features and scope strategies. After, we propose the design of EMA and validate it with a case study discussion.
Results. Using a concrete JavaScript implementation of EMA, named EMAjs, we can implement two Web applications: a smartphone application as an example to illustrate EMAjs in action, and an application of home automation to discuss and compare our proposal.
Conclusions. Our proposed mechanism allows developers to instantiate different activation scope strategies and interfaces to decouple the declaration of activation mechanism conditionals from the base code.

1. Introduction

Context-awareness is important for building pervasive systems as they have to adapt their behavior according to their current context of execution, gathered from the surrounding environment (*e.g.,* location, users' preferences) [1]. The Context-oriented Programming (COP) [2] paradigm allows developers to dynamically change the software behavior in response to an identified context. COP languages provide dynamic adaptations by means of *layers* that identify *contexts* [3] as concrete situations sensed from the surrounding environment, and execute *partial methods* [4] that are specialized fine-grained behavior of the system's base code.

A layer is activated or deactivated following a defined *activation mechanism*. Whenever the system identifies the execution of a context, the layer associated to the context activates, otherwise, it deactivates. A large number of activation mechanisms have been proposed in the literature [2,5–14], each, tailored to address a particular development need. Given the target to specific needs, these mechanisms seldom share the same semantics, for example, *event-based* mechanisms activate a layer when a pattern of triggered events happens. As a consequence, developers end up tweaking layer declarations in contortive ways, or creating new specialized activation mechanisms. We highlight two problems from this observation. First, each activation mechanism offers a single and fixed semantics to activate a layer (Fig. 1a). However, interaction between layers can be complex, requiring different mechanisms for their activation. To enable such interaction and increase the flexibility of the system, we require an activation and scoping model that unifies different mechanisms. For example, a global activation scope causes a smartphone to reduce the precision of calculations whenever its battery level is low (*i.e.,* the context). However, some

* Corresponding author.
E-mail address: pleger@ucn.cl (P. Leger).

https://doi.org/10.1016/j.infsof.2022.107132
Received 7 September 2022; Received in revised form 28 November 2022; Accepted 6 December 2022
Available online 13 December 2022

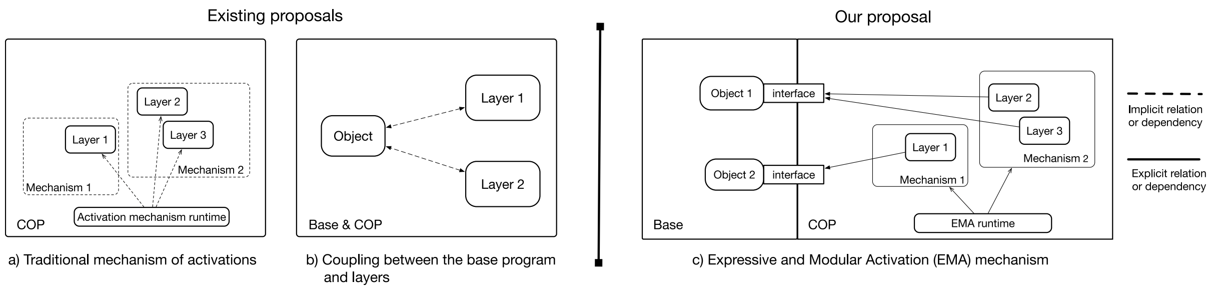

Existing proposals
Layer 2
Layer 1
Layer 3
Mechanism 2
Mechanism 1
Activation mechanism runtime
COP
Object
Layer 1
Layer 2
Base & COP
a) Traditional mechanism of activations
b) Coupling between the base program and layers
Our proposal
Object 1
interface
Object 2
interface
Layer 2
Layer 3
Mechanism 2
Layer 1
Mechanism 1
EMA runtime
Base
COP
c) Expressive and Modular Activation (EMA) mechanism
Implicit relation or dependency
Explicit relation or dependency

Fig. 1. Comparison between existing activation mechanisms and our proposal.

operations may require higher precision, and would benefit from a local activation scope to keep the precision for those instances. Second, most activation mechanisms keep a close relationship with the system's base code, implying a coupling between the layer declaration, its use, and the base code. This is so because developers implicitly depend on variable references or method invocations in the base code to declare a conditional (Fig. 1b), increasing the fragility of dynamically adaptive programs. Changes in base code or a conditional declaration may spuriously activate or deactivate layers [15]. A similar problem has been identified in other areas like aspect-oriented programming [16], known as the *fragile pointcut* problem [17].

To address the previous issues, we propose an EMA mechanism for COP languages. As Fig. 1c shows, EMA complements ideas to expressively declare customized activation mechanism scope strategies [18] with interfaces to define conditionals of these mechanisms that remove implicit dependencies with the base code [19].[1] EMA uses a unified activation mechanism interpreter that enables managing case by case activation scoping strategies by means of dedicated predicates to arbitrate interaction among layers. In addition, this language allows developers to specify interfaces to exhibit part of the internal system's state (*i.e.,* field values) and behavior (*i.e.,* method executions) of an object, using predicate conditionals to *explicitly* associate variable identifiers available in these interfaces. As a consequence, developers can define layers with low coupling to the base code, in which the layer itself can include the particular specification of its activation scope strategy. Our proposal works because these two complementary ideas help each other: scope strategies use interfaces to be defined, and interfaces with scope strategies expressed in another layer component make simpler conditional definitions. Finally, as a benefit for EMA developers, layers improve their reuse, flexibility, and their modularity to be applied in adaptive software systems. To highlight the novel of this proposal, we develop a reference frame that compares the existing proposals in terms of activation mechanisms and scope supported. In summary, this paper makes the following four contributions:

1. An Expressive and Modular layer Activation (EMA), which provides an expressive API for layers scoping, and interfaces between layers and the base code. With these features, EMA allows developers to enhance the modular definition of layers because their specification contains its scope and variable conditions to be (de)activated.
2. A concrete implementation in JavaScript, named EMAjs, with a preliminary performance evaluation.
3. A discussion of two applications that use our concrete implementation, where one is compared to object-oriented paradigm and context-oriented paradigm without EMAjs. Additionally, we discuss the threats to validate our proposal.
4. An extensive and updated reference frame that compares COP languages in terms of activation mechanisms and scope supported.

[1] Basic and immature notions of these two ideas have been presented independently and disconnected at the Context-Oriented Programming workshop.

The rest of this paper is organized as follows. We motivate the need of an expressive and modular layer activation mechanism using a prototypical example of a smartphone's adaptive display orientation application in Section 2. Based on this motivation, Section 3 presents the background of COP concepts required for our approach and a comparison of existing COP languages. Section 4 presents the idea and details of EMA, while its implementation details are made explicit by extending JavaScript with COP capabilities. Section 5 shows the validation through the implementation of two applications using EMAjs, performance evaluation, and threats to validate our proposal. Section 6 presents COP approaches related to our proposal, and finally, Section 7 closes the paper with the conclusion and avenues of future work.

***Availability*.** The current implementation of EMAjs is open-source and available at https://github.com/pragmaticslaboratory/EMAjs [20]. In addition, a running example presented in this paper is available at http://pleger.cl/sites/emajs (revision `903fdae`). Our proposal currently supports Nodejs (v16) [21], and the Google Chrome (v90) [22] and Mozilla Firefox (v88) [23] browsers, without the need for an extension.

2. Motivation: A mobile video game with an adaptive user interface

This section presents a variation of the context-aware smartphone display prototypical example [5,11,19]. We use a smartphone video game application with an adaptive layout interface with respect to the device's orientation. In COP, dynamic adaptations are carried out by means of software (re)composition as response to layer activation. This example motivates the difficulties and interaction intricacies between existing activation mechanisms, which supports the expression and modularity problems in activation mechanisms for COP languages.

The application offers the feature to adapt its layout and behavior to support the landscape and portrait orientations in such a way that the layout of the interface is always parallel to the ground, regardless of the physical orientation of the device. Fig. 2 shows the way the interface adapts by rearranging the distribution of displayed icons on the screen, in response to physical orientation changes, between *Portrait* and *Landscape*. As in the figure shows, an additional feature is *Keep Orientation*, which allows users to fix displayed components on the screen, even if the device rotates (*e.g.,* the video game scene). Implementations of such specialized features are *crosscutting concerns* [16] as they are scattered across many places in the base code, tangled with other concerns. Tangled features defining crosscutting concerns can be modularly implemented by means of COP abstractions.

Fig. 3 shows an excerpt the smartphone application with two classes. The `Screen` class captures the physical orientation and rotates window components in the application, and the `wComponent` class is in charge of displaying the application components using a `display()` method. The behavior of the application is specialized

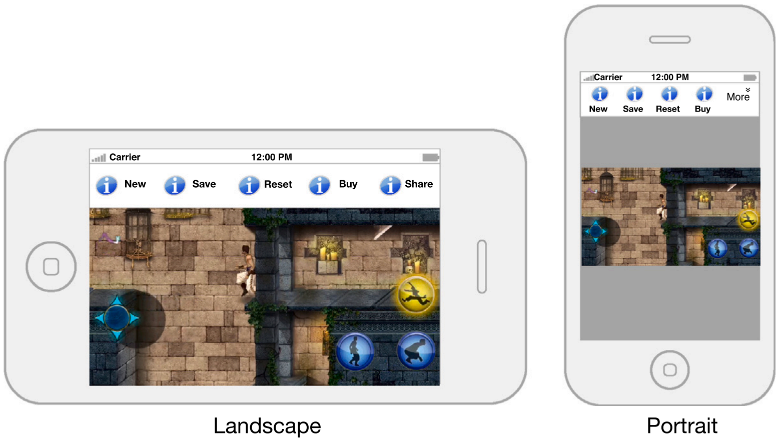


Fig. 2. A video game with the (almost) full adaptive mode.

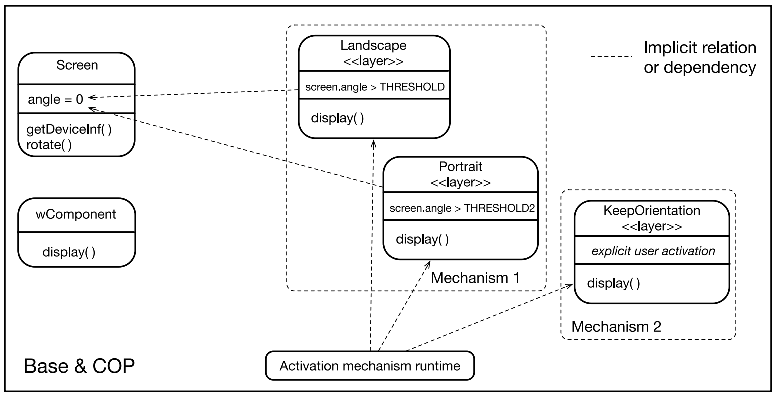


Fig. 3. Implicit relation and dependency between base code and layers with their activation mechanisms in the smartphone example.

by three layers: `Landscape`, `Portrait`, and `KeepOrientation`. `Landscape` displays window components using a landscape layout, for example, the icons in a menu bar with their name showing to the right. `Portrait` displays window components using a portrait layout, for example, the icons in a menu bar with their name underneath the icon. Finally, `KeepOrientation` keeps specific displayed components fixed on the screen, even if the device rotates (*e.g.,* the video game scene in Fig. 2). Note that the changes to the `display()` method to customize the way components appear on the screen crosscuts objects from different classes of the application (*e.g.,* icons, labels, scene).

Whereas previous layers can be implicitly activated when a variable changes its value, the `KeepOrientation` layer is explicitly activated on an object of a window component. Therefore, we have at least two different ways to deal with the activation of layers: global activation and per-instance (object) activation. In addition, triggering the dynamic adaptation of the behavior associated to each of the layers `Landscape` and `Portrait`, these layers need to access the `angle` variable from the `Screen` class, implying an implicit dependency between these two layers and the base code.

Summary. Taking into account the situation described for the smartphone display orientation example, we highlight two problems that arise from the existing activation mechanisms in COP languages:

1. Lack of an expressive activation mechanism scope strategy that allows developers to have customized and specialized scope semantics for a layer.
2. Coupling between layers and base code with implicit dependencies, making fragile programs that use COP.

As a consequence of these previous two problems, the reuse, flexibility, and modularity of layers are hindered. To the best of our knowledge, existing COP languages do not allow developers to customize an expressive scope strategy together with decoupling layers and the base code. As our proposal does, previous both features put together make it possible to define full-fledged modularized layers.

3. Context-Oriented Programming

COP [2] allows programmers to develop software systems that adapt their behavior at runtime according to identified contexts from the surrounding execution environment. The implementation of behavior adaptations, together with their associated layers and their activation, crosscut several concerns of the system. Before diving into our proposed approach in the next section, this section introduces the main concepts of COP, and discusses the existing activation mechanisms and scoping strategies. Finally, this section discusses different COP languages in terms of the two previous features.

3.1. Layers

The main abstraction in COP languages is a *layer* [4],[2] which encapsules the components to modularly implement the dynamic adaptation of base code behavior. This dynamic adaptation is carried out by

[2] Layers are sometimes refer to as contexts [5]. Following most of works (discussed in this paper), we will use the term *layers* as language abstractions for dynamic adaptations that are related to an identified *context*.

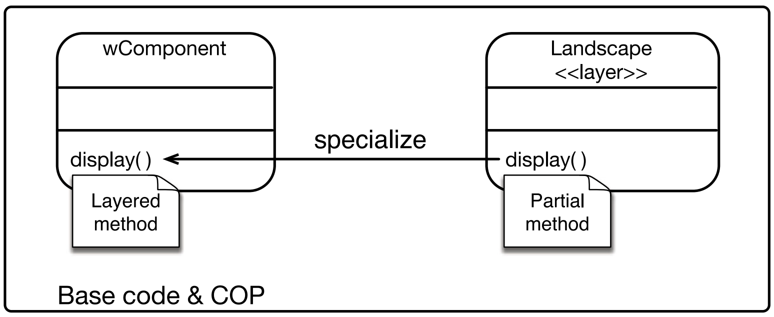


Fig. 4. Layer method and its partial method.

partial method [4] definitions, which specialize method implementations (known as *layered methods*) in object-oriented languages (Fig. 4). Snippet 1 illustrates the definition of a layer with a partial method in the smartphone application example using ServalCJ [11,14], a Java-like for COP. In ServalCJ, developers can define layers (Lines 1–3). Within a class declaration, ServalCJ developers can express how a layer modifies the class behavior (Lines 10–15). In this case, when the `Landscape` layer is activated, the `display()` behavior will be the one declared in the method inside the layer (Lines 11–13), rather than the base definition in the `wComponent` class (Lines 6–8). In COP languages, a partial method can execute the original implementation of its layered method using a construct like `proceed`.

```
1  layer Landscape {
2    //other code
3  }

5  class wComponent {
6    void display() {
7      //implementation
8    }

10   layer Landscape {
11     void display() {
12       //variation for the display method
13     }
14   }
15 }
```

Snippet 1: Partial method and layer definition in ServalCJ

3.2. Layer activation mechanisms

As a layer is dynamically activated and deactivated, its associated partial methods are respectively composed with and withdrawn from the system. Activation mechanisms in layers establish how and when a layer must be activated. We can find different kinds of mechanisms to activate layers: *Imperative* mechanisms [2,5] activate layers explicitly in the system execution workflow, using constructs as `with` or `activate`, *implicit* mechanisms [8,10–12,24,25] activate layers whenever a specified condition is satisfied, and *event-based* mechanisms [9,26] use event matching triggers in combination with imperative and implicit mechanisms [13,14]. Using the smartphone application example, Fig. 5 illustrates the three mechanisms to activate the `Landscape` layer when the angle of rotation for the smartphone exceeds a threshold. Unlike the imperative activation mechanism, other activation mechanisms use a conditional that must be satisfied to activate a layer, as Snippet 2 describes. The snippet shows the manual management of relations between layers. For example, the interaction between `Portrait` and `Landscape` must explicitly deactivate the opposite layer whenever one of the two layers is activated. To resolve relations and activation conflicts between layers, COP languages add *ad hoc* constructs to existing activation mechanisms.

Snippet 2: Layer activation interaction management

```
activate Landscape if(Screen.angle > THRESHOLD);
activate Portrait when !Landscape;
```

3.3. Layer activation scope strategies

We differentiate *when a layer is active* from *when a layer has to be applied*. The last is the scope strategy for an activated layer, meaning that the layer may keep active although its scope does not apply in a certain portion of a program execution. This difference can be useful for several examples when we need:

1. a *persistent layer* which keeps information in an active layer and not applied, *e.g.,* the rotation angle that activated the landscape layer.
2. not apply an active layer in some particular place without an execution of the enter and exit transition process, *e.g.,* the KeepOrientation layer.
3. to apply a layer to specific objects that could be created under some conditions, *e.g.,* objects created by untrusted third applications inside of the video game like advertisements.
4. to apply a layer considering the relation with other (de)active layers, *e.g.,* Landscape and Portrait layers.
5. any combination of previous ones, *e.g.,* our proposal.

Researchers have already discussed different scope strategies (combinations of lexical and dynamic scope) in programming languages [27], and particularly in areas related to COP [28–30]. Specifically in COP, ContextJS [31] allows developers to express some scope strategies to a layer (more in the next section).

3.4. Context-oriented languages

In the body of literature, we can find several COP languages, starting by ContextL [4] which is one of the first COP language. Ordered by the release year, this section briefly describes COP languages in terms of activation mechanisms and scope strategies (summarized in Table 1).

ContextL. In 2005, Costanza and Hirschfeld proposed ContextL [4] as an extension of the Common Lisp Object System (CLOS) [42] to support context-oriented programming. ContextL uses an explicit activation mechanism utilizing the construct `with-active-layers`, and layers are dynamically scoped to a specific control flow of a code block.

Ambience. González et al. propose Ambience [34], an implementation of the AmOS [43] object system to support COP. Although the language uses the term *context* to refer to a *layer*, we can find the same conceptual abstractions that the rest of COP languages. Similar to ContextL, the language uses and an explicit activation mechanism, and layers are dynamically scoped to a specific control flow and global.

ContextJ. A Java extension to support COP is ContextJ [2,26]. The language uses an explicit activation mechanism through the `with`

```
//imperative (contextJ)

if (Screen.angle > THRESHOLD) {
   with(Landscape) {
    //base code
    wComponent.display();
  }
}
```

```
//event-based (JCop)

on(* wComponent.display()) &&
when (Screen.angle > THRESHOLD) {
     with(Landscape);
}
```

```
//implicit (ServalCJ)

activate Landscape
    if (Screen.angle > THRESHOLD);
```

Fig. 5. Activating the `Landscape` layer using three different activation mechanisms.

Table 1
Activation mechanism and scope strategies of existing COP languages.

Year	COP Language	Language	Activation mechanism	Scope strategies
2005	ContextL [4]	CLOS	Explicit	Control flow
2007	ContextS [32]	Squeak	Explicit	Control flow
2007	PyContext [33]	Python	Implicit	Control flow
2007	Ambience [34]	CLOS-like	Explicit	Control flow & global
2008	ContextJ [2]	Java	Explicit	Control flow
2009	NextEJ [35]	Java-like	Explicit	Control flow
2010	ContextErlang [36]	Erlang	Explicit	Per-actor
2010	JCOP [26] (ContextJ variation)	Java	Event-Based	Control flow
2010	ContextPy [37]	Python	Explicit	Control flow & global
2010	ContextLua [38]	Lua	Explicit	Control flow
2010	Subjective-C [5]	Objective-C	Explicit	Global & per-thread
2011	EventCJ [9]	Spoofax/IMP	Event-Based	Per-object
2011	ContextJS [31]	JavaScript	Explicit	Control flow & global & per-object (customizable)
2013	ServalCJ [39] (or Javanese)	Java-like	Implicit	Per-object & global
2013	L [40]	Java-like	Explicit	Control flow
2013	Context Traits [6]	JavaScript	Explicit	Global & per-object
2015	Congolo [41]	Golo	Implicit	Module
2017	ServalCJ extension [11]	Java-like	Implicit	Per-object & global
2018	Emfrp extension [12]	Emfrp	Implicit	Global

construct and follows the scope of a control flow of dynamic extent. A ContextJ variant is JCop [26], which uses join points and pointcuts from AspectJ [44] to support an event-based activation mechanism.

Subjective-C. An extension of Objective-C is proposed to enable context-orientation for mobile devices [5]. Subjective-C acts as a pre-compiler on top of Objective-C using metaprogramming to dynamically modify method implementations during method dispatch. In Subjective-C adaptations use an explicit (de)activation mechanism (with dedicated @activate and @deactivate abstractions), immediately enacting such (de)activation. In Subjective-C adaptations are scoped both globally and locally to all objects in an execution thread. Moreover, Subjective-C was the first language to reconcile multiple scoping mechanisms in a single (de)activation model [7].

EventCJ. This is a compiler that is built on the top of Spoofax/IMP language workbench [45]. EventCJ [9] translates developer pieces of code to AspectJ and Java. The compiler enables to developers to use an event-based activation mechanism using the matching of join points. Additionally, EventCJ layers have can define a per-object scope instead of the dynamic extent of a specific control flow like ContextJ.

PyContext. A Python extension to support COP [33]. PyContext supports an implicit activation mechanism that verifies which layers are active before a layered method is executed. Similarly ContextJ, PyContext uses a scope that follows a specific control flow.

ContextJS. Lincke et al. propose ContextJS [31], a JavaScript extension for COP. The proposal uses an explicit activation mechanism. Regarding the scope strategies, ContextJS offers an open customization for scope strategies that allow developers to express dynamic extent of a block code, per-object with restrictions, and global.

ContextErlang. An Erlang extension to support COP [36] in distributed and concurrent environments. ContextErlang is used to develop self-adaptive applications that based actor-oriented programming. The extension uses an explicit activation mechanism which is scoped by actors.

ServalCJ. Kamina et al. propose ServalCJ [39], a Java-like language that supports COP. Similar to PyContext, the language supports an implicit activation mechanism that activates layers when conditionals are satisfied. In [11], ServalCJ is extended to support an implementation variant of implicit activation that uses reactive values to activate layers. This variant allows a layer to be (de)activated immediately when its associated conditional changes. ServalCJ supports two scope strategies: per-object and global layer activations.

Context Traits. Context Traits [6] posits a compositional way to enable dynamic variations to the contexts based on the concepts of traits and trait (re)composition. This language uses an explicit activation mechanism in which the (de)activations take place immediately after they are called. Similar to our approach, enables de adaptation of specific object instances, or all objects of a class, depending on the association of behavior variations with object instances or their class representative.

Congolo. Golo [46] is a lightweight dynamic language for the Java Virtual Machine (JVM). Congolo [41] is a COP extension for Golo, which uses an implicit activation mechanism, the extension follows the per-module scope strategy.

Emfrp extension. Emfrp is a pure-FRP (Functional Reactive Programming) language for small-scale embedded systems [47]. In [12], Watanabe proposes a COP extension for Emfrp. Similar to ServalCJ, the extension uses an implicit activation mechanism and supports a global scope.

COP languages like ContextS [32], ContextPy [37], L [40], NextEJ [35], and ContextLua [38] (mainly) propose to use an explicit activation mechanism and its scope follows the control flow or a global layer activation. In addition, we can find other proposals like Lambic [24] and Flute [25], proposed COP language introducing the definition and realization of adaptations to the context for group behavior and reactive programming, respectively. Finally, we can find alternatives to address issues related to context-oriented implementations that go beyond the concepts of layers like the language Korz [48] and context-oriented behavioral programming [49]; however, these proposals are beyond the scope of this paper.

4. EMA

This section presents EMA, an expressive and modular activation mechanism for COP. Our proposal complements ideas to expressively declare customized activation mechanism [18] with interfaces to define conditionals of these mechanisms that remove implicit dependencies with the base code [19]. Using the smartphone application example, we describe how each idea to work together in EMA as a complementary manner.

4.1. An expressive API for scoping

To manage layer activation scope, we define an expressive API (Application Programming Interface) that enables developers to activate/deactivate multiple layers simultaneously, defining dedicated and customized scoping strategies for each particular activation.

Layer (de)activation is performed using our defined API to explicitly and expressively state the scope in which a layer is active. As an example of the need for this scope in activation mechanisms, consider the activation of the Landscape layer in our video game application, defined in Section 3.2 using ServalCJ as `activate` Landscape `if`(Screen.angle > THRESHOLD). If such conditional is not satisfied, the layer is deactivated. Such activation mechanisms present two problems.

First, the activation mechanism explicitly targets a specific layer. For example, each activation and deactivation of the Landscape and Portrait layers in the video game application must be managed independently, even though they refer to the same system property. This hinders the conciseness of activations, opposed to implicit or more flexible mechanisms. Second, scoping rules in activation mechanisms are managed uniformly. However, different layers may require different scopes to assure the correct system behavior. For example, in the video game application, the Landscape layer can be activated with a global scope, applying the specialized behavior to all entities in the system, while the KeepOrientation layer may be applied to specific objects (*e.g.,* instances of wComponent) for which we do not want to change the orientation along with the device.

To the best of our knowledge, the purpose of our expressive scoping API within EMA arises from the observation that there is no unifying interface for scoping the activation/deactivation of layers that is independent from a COP language. We address both problems existing in common activation mechanisms in COP. The idea behind our API is to enable layer (de)activation through the use of the interface to which layers subscribe (Section 4.2). The activation scopes are defined as functions on layers' interfaces, which are able to apply the action to multiple layers simultaneously.

The expressive scoping API for layer activation and deactivation used in EMA is defined in Table 2. Each function specifies the action (`activate` or `deactivate`) to execute over a set of layers specified by a predicate over a conditional (predicate) on conditions over interfaces, and applying the activation/deactivation on a particular scope.

The activation and deactivation functions effectively actuate over all layers satisfying the condition predicate on the `interface`. By default,

Table 2
Flexible layer activation/deactivation API.

LayerActivation	::=	`activate`(*PredicateExpression* [, *Scope*])
LayerDeactivation	::=	`deactivate`(*PredicateExpression* [, *Scope*])
PredicateExpression	::=	{*Predicate*}(*Condition*)
Predicate	::=	`between` | `allOf` | `atLeastOne` | `atMostOne` | `unique`
Condition	::=	*Range* | *ConditionPredicate*
Scope	::=	*Instance* | *Class* | *Thread*

all layer activations and deactivations use a global scope. *ConditionPredicates* (*e.g.,* RotationCondition) specify a predicate that the `interface` must satisfy to enact the required action. For example, `activate`(allOf(RotationCondition)) results in the activation of the layer (Landscape) for all the program entities that satisfies the specified *ConditionPredicate*:

```
RotationCondition = () => Screen.angle > THRESHOLD
```

The predicates atLeastOne, atMostOne respectively activate any (possibly several), and at most one, of the layers satisfying the given *Condition*. The unique predicate lets us express the condition where just the specified layer satisfying the condition must be active (all other layers are implicitly deactivated). In the example in Snippet 2, assuring the activation of Portrait takes place when Landscape is inactive, can be expressed as `activate`(unique(RotationCondition)). Finally, the between predicate enables the activation of all layers for which the *Condition* is satisfied, specified as a *Range* query. In addition to *PredicateExpressions*, each action can specify the *Scope* of the activation/deactivation for the layers, following any of the existing five models:

(1) local to the lexical scope in which the adaptation is defined/-called [4] (using withCurrentScope),
(2) local to all the objects in the current execution thread [7] (using a specific *thread* id),
(3) global to all objects and all execution threads [50] (default behavior not specifying the *Scope*),
(4) local to a specific object instance [9] (specifying an object *Instance* as *Scope*), and
(5) specific to all object instances of a type [51] (specifying a *Class* as *Scope*).

As a consequence, EMA eases the layer activation scope from two perspectives. First, we provide a unified interface to scope adaptations using different mechanisms within the same application. Second, we enable the selection of many layers based on their interfaces.

4.2. Interfaces

Fig. 6a shows the implicit dependency between base programs and layers through program variables used in conditional declarations to activate layers, which is problematic for code maintenance and reuse. EMA uses interfaces [19] to decouple such implicit dependency. Taking inspiration from Open Modules [52] and other related proposals [53–55], interfaces allow developers to exhibit field values and method executions (with a return value) from any object of a class. Note that exhibiting a method without a return value would not make sense to define a *Condition*, as conditions are defined for base-level values; therefore that is not allowed. Exhibited values are used in the *ConditionPredicate* declaration to activate and deactivate layers. For example, Fig. 6b shows the Screen class exhibiting the angle variable as a rotation interface (shown in the white square in the figure), to satisfy the conditional requirements of the Landscape layer. Interfaces also accept expressions as values to address the problem of context reuse. For example, consider a new device (*e.g.,* tablet) whose rotation angle is provided in radians. As the layer requires the rotation in degrees, the expression rotation: angleInRadian*180/PI is used in the interface definition

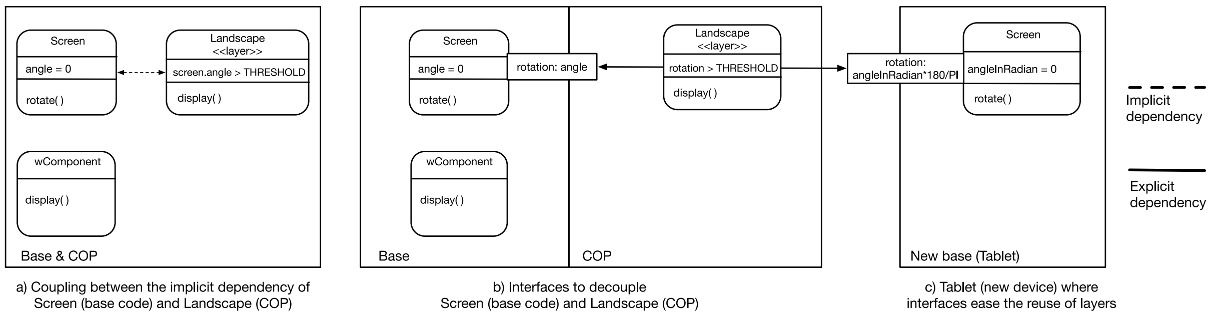

Screen
angle = 0
rotate( )
Landscape
<<layer>>
screen.angle > THRESHOLD
display( )
wComponent
display( )
Base & COP
a) Coupling between the implicit dependency of Screen (base code) and Landscape (COP)
rotation: angle
rotation > THRESHOLD
Base
COP
b) Interfaces to decouple Screen (base code) and Landscape (COP)
rotation: angleInRadian*180/PI
angleInRadian = 0
New base (Tablet)
c) Tablet (new device) where interfaces ease the reuse of layers
Implicit dependency
Explicit dependency

Fig. 6. Coupling due to the implicit dependency between base program and layers, and how interfaces can decouple them.

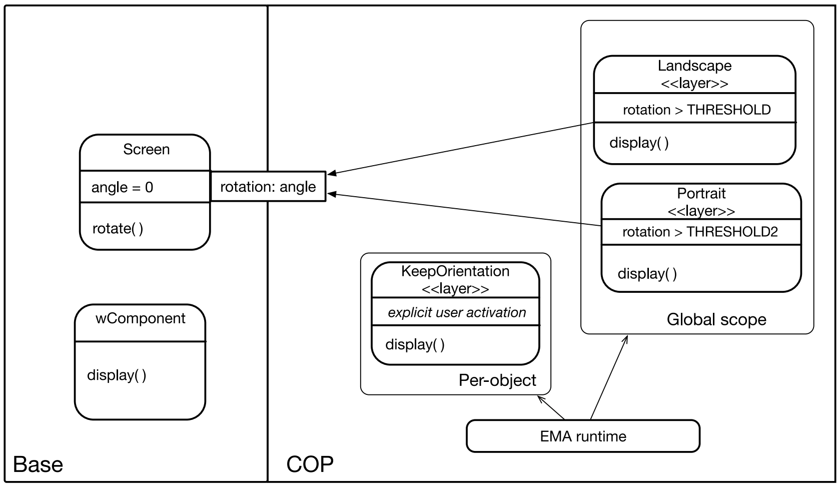


Fig. 7. Application of EMA to the smartphone example.

to reconcile both implementations, as shown in Fig. 6c. As a layer is an object, its conditional evaluation can be exhibited to other layers to resolve activation conflicts. Through interfaces, developers can replace the implicit dependency between layer definition and activation, and the base program.

4.3. Complementing an expressive scoping API and interfaces

EMA, as an expressive and modular activation mechanism, allows developers to define customized activation mechanisms according to their needs. This proposal is carried out by putting together an expressive API for scope strategies and interfaces, in which developers can expressively and modularly define layers. Fig. 7 shows the application of EMA for the case of the smartphone video game. First, we can see that developers can customize different scope strategies to layers: Landscape and Portrait use a global scope to modify window components, and KeepOrientation uses a per-object activation scope (*i.e.,* video game scene). Second, the Screen class exhibits the angle field as a rotation interface to satisfy the requirements for the Landscape and Portrait layers. In this way, the layer definitions are now decoupled from the smartphone application, *i.e.,* the base program. Notice that both scope API and interfaces are powered when they cooperate with each other. This is because the scope API are defined as interfaces on layers, and this API allows developers to remove scope restrictions from the conditional definitions used by interfaces to (de)activate a layer.

4.4. EMAjs

This section presents a concrete implementation of EMA as a JavaScript library, named EMAjs [20].[3] Readers should note that while our implementation uses the object-oriented paradigm, the model is agnostic to a concrete paradigm could be implemented, for example, using functions as first-class entities.

In EMAjs, we introduce two main abstractions to increase the expressiveness and modularity of layer activation: interfaces, and dedicated scoping rules.

Interfaces are implemented using a reactive programming style [19]. Interfaces are created using the exhibit language abstraction. All interfaces are registered and kept in a signalInterfacePool as part of a management object, EMA. Upon creation, we associate the new interface with the layers deployed in the environment, as a condition for the layer activation/deactivation. Layer activation conditions, coming from interfaces, are created as reactive signals, so that whenever the interface value changes, the layer condition is evaluated reactively. If the condition is satisfied, the layer is activated, otherwise the layer becomes inactive.

In addition to the layer activation/deactivation using interfaces, we extend the scoping of layers following a declarative set of rules implementing different predicates applicable to layers and the program

[3] The EMAjs implementation is available at: https://github.com/pragmaticslaboratory/EMAjs.

entities they affect [18]. Scoping rules can be specified by developers upon the explicit activation/deactivation of a layer, using the `activate` /`deactivate` abstractions. At layer activation/deactivation developers can specify the activation conditions and restrictions, the extend of the layers' effect, and the entities over which the layer is applied, as explained in Section 4.1. To do this, the EMA manager filters the application of the layer to the application entities based on the given predicate to the activation, applying the layers exclusively for the entity instances specified in the activation conditions and for the specified extend (*i.e.,* global or local).

These two additions to COP languages, together with the definition of partial methods, helps us define dynamic adaptations to the context that are more expressive and modular, as we explain in the following section.

5. Validation

We now present the validation of our work evaluating the modularity and flexibly benefits of the features to scope adaptations introduced in EMA. To do this we use the EMAjs implementation on two COP systems, coming from different applications domains. Additionally, we compliment this evaluation of the new features in EMA with a performance evaluation, contrasting EMA to implementations using layers and objects, to gauge the speed-up incurred by implementing interfaces. Finally, we put in perspective the results of our validation with a discussion of the limitations and threats to validity of our approach.

5.1. EMA application

We validate the features introduced in EMA by means of two applications from different domains and sizes, these are: a smartphone video game application, and a smart-home environment.

The smartphone application is used to show case the main features introduced with EMA, in a small scale application. Additionally, the contexts in this application are characterized as internal to the application — that is, they are monitored variables from the system objects. The smart home application is a larger scale application in which context changes are driven by variables external to the system gathered through sensors. With this application we evaluate further EMA's features with respect to the modularity and flexibility of COP applications by comparing it with alternative implementations using other COP languages (*i.e.,* without EMA), and object-oriented implementations.

Overall, our validation shows that using EMA in context-aware applications leads to solutions that are more expressive and modular than their counterparts using other COP languages.

5.1.1. Smartphone video game application

The first part of the validation consists of a description of the smartphone vide game application implementation using EMAjs,[4] to showcase the main components of EMA at work.

First, system objects may use the exhibit construct to define interfaces. This construct takes an object and creates an interface from specified object properties (both fields or methods with return values). Snippet 3 shows the definition of the rotation interface, taken from the angle field in the Screen prototype object.

```
EMA.exhibit(Screen, {rotation: "angle"});
```

Snippet 3: Defining and exhibiting an interface

Second, we add a Condition as a function object providing a user-defined predicate that specifies the situations in which a layer should become active. Condition predicates should be defined in terms of an interface to capture a particular situation from the system state. Layers define a conditional property listing all the condition entities that may be applicable to the layer. Snippet 4 shows the definition of the Landscape layer (Lines 1–6) and a RotationCondition object (Line 8). Similarly the ServalCJ extension [11], layers may also define two optional methods, to execute predefine actions as soon as the layer is activated (onEnter()), and as soon as the layer is deactivated (onExit()). Partial method definitions are associated with the layers in which they apply by means of the addPartialMethod construct. This construct takes a layer, a prototype object, a method selector, and its partial implementation, to associate the partial implementation to the method selector in the prototype. In our case, Lines 11–13 define the behavior variation to the display method of the wComponent prototype object. Finally, developers use the EMA.`layer` method to deploy a layer (Line 14).

```
1  let Landscape = {
2    conditional: [RotationCondition],
3    //optional methods
4    onEnter: function() {Screen.rotate();},
5    onExit: function() {...}
6  };

8  let RotationCondition = ()=> rotation > THRESHOLD;

10 EMA.addPartialMethod(Landscape,
11     wComponent,display,function() {
12     //method variation
13 });
14 EMA.layer(Landscape);
```

Snippet 4: Defining and exhibiting an interface

Finally, the activation and scoping mechanisms of layers and their associated partial methods become available to the runtime system by means of the new `activate` and `deactivate` constructs. Snippet 5 shows some of the different mechanisms, available in EMA, to scope the activation and deactivation of layers. Line 1 uses a global scoping, in which all layers satisfying the RotationCondition predicate become active. Line 2 also uses an activation with global scoping, where the unique predicate assures that only the layers that satisfy the RotationCondition are active, implicitly deactivating all other layers. Line 3 uses a scoping rule in which all layers satisfying FixedCondition become active exclusively for the gameScreen object instance, while other objects are unaffected by the layer. Line 4 uses a scoping rule in which, at least, one of the layers satisfying RotationCondition deactivates from a specific execution thread.

```
1 EMA.activate(RotationCondition);
2 EMA.activate(unique(RotationCondition));
3 EMA.activate(allOf(FixedCondition), gameScreen);
4 EMA.deactivate(atLeastOne(RotationCondition), thread);
```

Snippet 5: Scoping mechanisms for layer (de)activation

From the implementation of the smartphone application, in Snippet 6, we note two characteristics that help us to support the statements of this paper when compared to implementations in other COP languages:

1. We note the decoupling between the base behavior, layers, and partial method definitions. The use of interfaces allows us to define layers and partial methods as modules independent of base code, *i.e.,* class definitions. In the Screen class (Line 1 in Snippet 6), we can see the class declaration is independent from the declarations for the Landscape and KeepOrientation layers (Line 10). Approaches as that of ServalCJ follow a layer-in-class declaration [56] as in Snippet 1. Additionally, interfaces prevent the dependency between base-level variables and layer activation conditions (Line 6 in Snippet 6), increasing its modularity. A second point of decoupling takes place between the definition of partial methods and layers. We define partial methods outside of layers, unlike other COP languages, to let base programmers define the specialization of their system's behavior for any layer, and not just the ones defined by them. The

[4] The full running implementation is available at: http://pleger.cl/sites/emajs.

addPartialMethod method lets developers associate: a layer, the class on which it takes effect, and the specialized behavior (Line 17).

2. EMAjs allows developers to define different scoping mechanism to take effect in each activation instance of a layer (Line 28). Effectively, this implies that developers are able to express individual layer activations with different scoping mechanisms at different execution points. As consequence, layer activation is more flexible, allowing for richer interaction between adaptations.

```
1  let Screen = {
2    angle: getDeviceInf().orientation,
3    rotate: function() {/* implementation */}
4  }

6  EMA.exhibit(Screen, {rotation: "angle"});
7  let RotationCondition = ()=> rotation > THRESHOLD;
8  let FixedCondition = ()=> rotation == "fixed";

10 let Landscape = {
11   condition:()=> RotationCondition;
12 }
13 let KeepOrientation = {
14   condition:()=> FixedCondition;
15 }

17 EMA.addPartialMethod(Landscape, wComponent, 'display', function()
       {
18     ('#grid').w2grid({
19        columns: [
20        { icon: 'icon1'},
21        { caption: 'new'},
22        // ...
23        { icon: 'icon5'},
24        { caption: 'share'},
25      ]});
26 });
27 let gameScreen = new wComponent();
28 EMA.activate(unique(RotationCondition));
29 EMA.activate(allOf(FixedCondition),gameScreen);

31 EMA.layer(Landscape);
32 EMA.layer(KeepOrientation);
```

Snippet 6: Video game application in EMAjs

5.1.2. Home automation system

We use a home automation system as a case study to evaluate the modularity and flexibility of EMA. The home consists of different rooms and appliances within those rooms that users can interact with. For this case study we focus on the behavior of door ring alerts.

The base behavior of the alert system is to advertise visitors at the front door of the home with a ringtone. The base alert sounds in a standard chime installed at the door to provide visitors feedback about the action. To advertise the alert to home inhabitants, the ringtone should echo throughout the home. However, such alerts should echo only in the rooms occupied by a user; which is defined as an adaptation to the regular behavior. Whenever a user enters a room, the ring alert is forwarded to the appliances in the room (*e.g.,* TV, radio, soundbar) by echoing the sound. A second adaptation is defined to manage alerts according to the appliances currently in use. Alerts should not disrupt the regular behavior of appliances, and therefore alternative notification options are used. For example, in the case of a TV, the alert is displayed as a text message on the screen rather than as sound. The home should also make sure that ring alerts do not echo in babys' rooms, not to disturb them. Moreover, if all rooms are occupied, the alert should sound on a single device, to avoid loud disturbances.

The purpose of this case study is to highlight the advantages of the activation and scoping mechanisms proposed in EMA with respect to the modularity and flexibility of adaptations and their activation. For this purpose we follow the evaluation strategy from the literature [57], where we compare the implementation of three versions of the case study. The first version implements adaptation definition through the use of the strategy design pattern. Adaptation activation is made explicit with manual global variables' manipulation, scoped globally. The second version implements a standard COP definition of adaptations with explicit activation and global scoping. The third version implements the system using EMA to manage both activation and scoping. Activation is implicit using exposed interfaces, and the scoping is tailored to each specific situation.

We compare the expressiveness and flexibility of the three adaptation models used in terms of the LOC required to define the modularity for three aspects for adaptation (*i.e.,* base system logic, adaptation logic, and activation logic) in terms of their scattering and tangling in the system.

To evaluate the modularity of EMA, we first measure how expressive is the language. We do this by measuring the LOCs for our case study, taking into account the special situations described before. Our hypothesis is that, if EMA is more expressive, then it should require a lower implementation effort (*i.e.,* LOCs) to develop the application concerns. Moreover, as EMA improves on the definition and management of dynamic adaptations, there should be a reduction in the percentage of code dedicated to these concerns. Fig. 8 shows the results of this measurement. The results confirm our hypothesis, that a more expressive approach reduces the total LOCs in the application. This reduction comes from the effort reduction (*i.e.,* LOC) in the adaptation logic and adaptation management concerns. This way developers can focus on the core logic of their application, rather than in the adaptation concerns. The EMAjs version of the case study is $0.21x$ more concise than its counter part using design patterns. When evaluating each of the concerns in the application, we observe that we reduce the adaptation logic in $0.17x$ and the adaptation management logic in $1.98x$, yielding an increase in the application logic of $1.38x$. Note that the increase in the application logic does not imply an increase in the LOCs for this concern, rather it highlights the importance of this concern in the application.

To continue the evaluation of the expressiveness of the applications, we turn our attention to the system modularity. Fig. 9 shows a division of the modules required to develop the application. Each of the modules represent a specific entity in the case study (in the x-axis) and the quantity of concerns that are defined/used within the module (y-axis). From Fig. 9, we observe that as we move towards more expressive languages, the complete system becomes more cleanly modularized, reducing the number of concerns used per module (*i.e.,* object file). Note that there is still some modules with multiple concerns, these correspond to the definition of layers, and the orchestration for the whole system. In the former case, we could split the definition of layers into individual modules, however this would increase the number of files. Given that the application is small, we preferred keeping all the definitions to a file. In the latter case, the concerns could also be split into different modules. This division depends on the characteristics and controls of the systems (*e.g.,* sensors) on the edge of the application. We note that in most cases for small application, at least one file will orchestrate the complete system, and therefore, containing all concerns.

Finally, Listings 7–9 complement our evaluation on modularity, in this case showing the behavior and corresponding adaptations for the situations when both a regular user and a baby are in a room.[5] Note that the color scheme in the listings follows the division of Fig. 8 into application logic, adaptation logic, and adaptation management. The dotted lines in the listings represent different application modules. From the code snippets it is possible to see how the code becomes more modular, by cleanly separating the three different concerns. First, we

[5] The purpose of the code snippets is to show the interplay and tangling between the three adaptation aspects, rather than to focus on the implementation details. The full implementation of the application is available at: https://github.com/pragmaticslaboratory/EMAjs [20].

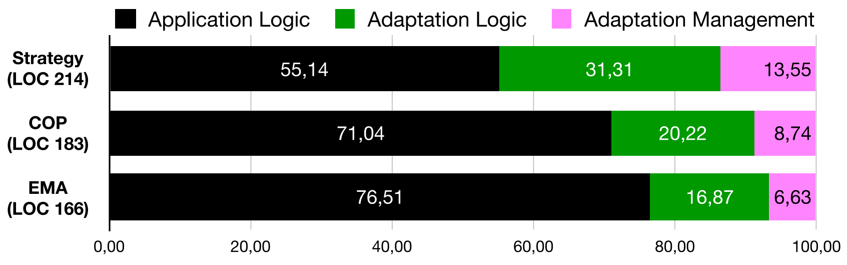


Fig. 8. LOCs percentage per application concern.

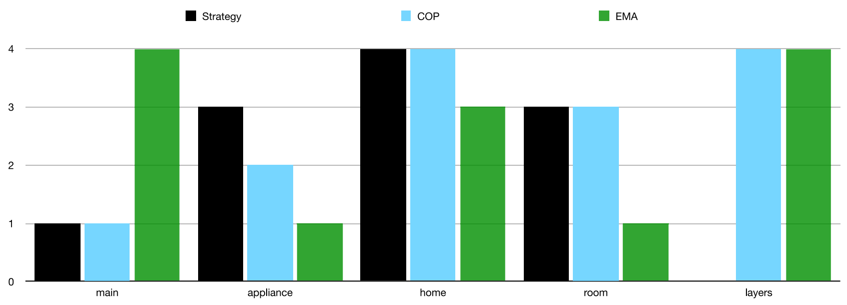


Fig. 9. LOCs percentage per application concern.

gain modularity by separating the adaptation logic from the base logic, when moving from the strategy pattern based implementation to an implementation using COP. However, in the latter implementation, we still observe, adaptation managing code tangled with the application logic. With EMA, we can remove that last dependency by exhibiting the appropriate interfaces, rather than by depending on each specific variable in the code.

Snippet 7: Pattern-based implementation of the case study

```
function Room(name, appliances) {
 this.strategy = null;
 this.users = 0;
 this. baby = false;
 this.userExit = function() {
   this.users = Math.max(0, this.users-1);
   if(this.users === 0)
     this.strategy = null;
   else
     this.strategy = new OccupiedStrategy(this);
 };
 this.babyInRoom = function() {
   this.baby = true;
   this.userEnter();
   this.strategy = new BabyRoomStrategy(this);
 };
}

function OccupiedStrategy(p) {
  Room.call(p.name, p.appliances);
  this.playSound = function() {
    p.appliances.forEach(a => {
      a.playSound("Advertise");
      console.log("ring alarm on");
    })
  }
}
OccupiedStrategy.prototype=new Room;

function BabyRoomStrategy(p) {
 Room.call(p.name, p.appliances);
 this.playSound = function() {
   console.log("Not playing a sound as the baby sleeps");
 }
}
BabyRoomStrategy.prototype = new Room
```

Snippet 8: COP implementation of the case study

```
let Room = {
  users: 0,
  baby: false,
  userExit: function() {
    this.users = Math.max(0, this.users - 1);
    if(this.users === 0)
      EMA.deactivate(OccupiedLayer);
    else
      EMA.activate(OccupiedLayer);
  },
  babyInRoom: function() {
    this.baby = true;
    this.userEnter();
    EMA.activate(BabyRoomLayer);
  }
}
-------------------------------------
EMA.addPartialMethod(OccupiedLayer, Home.rooms, "playSound",
  function() {
      this.appliances.forEach(a => {
        a.playSound("Advertise");
        console.log("ring alarm on");
      })
  }
);

EMA.addPartialMethod(BabyRoomLayer, Home.rooms, "playSound",
  function() {
    if(this.baby)
      console.log("Not playing a sound as the baby sleeps");
  }
);
-------------------------------------
let OccupiedLayer = {name:"occupied", condition: "", enter:
     function() {}, exit: function() {}};
EMA.deploy(OccupiedLayer);
let BabyRoomLayer = {name: "babyRoom", condition: "", enter:
     function() {}, exit: function() {}};
EMA.deploy(BabyRoomLayer);
```

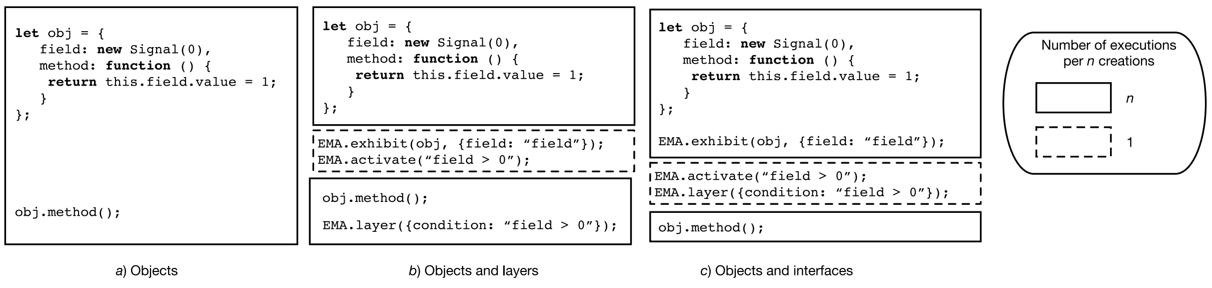
Snippet 9: EMAjs implementation of the case study

```
let Room = {
  users: new Signal(0),
  baby: new Signal(0),
  userExit: function() {
    this.users.value = Math.max(0, this.users.value - 1);
    if(this.users.value === 0)
      this.fullHome.value = Math.max(0,this.fullHome.value-1);
  },
  babyInRoom: function() {
    this.baby.value += 1;
  }
}

EMA.exhibit(Room, {occupied: Room.users});
EMA.exhibit(Room, {withBaby: Room.baby});
-------------------------------------
EMA.addPartialMethod(OccupiedLayer, Home.rooms, "playSound",
  function() {
    this.appliances.forEach(a => {
      a.playSound("Advertise");
      console.log("ring alarm on");
    })
  }
);

EMA.addPartialMethod(BabyRoomLayer, Home.rooms, "playSound",
  function() {
    if(this.baby)
      console.log("Not playing a sound as the baby sleeps");
  }
);
-------------------------------------
let OccupiedLayer = {name:"occupied", condition: "occupied > 0",
     enter: function() {}, exit: function() {}};
BabyCondition = "withBaby > 0"
let BabyRoomLayer = {name: "silent", condition: BabyCondition,
     enter: function() {}, exit: function() {}};
```

To evaluate the flexibility of EMA, we focus our attention in the interaction between the OccupiedRoom and BabyRoom adaptations. The OccupiedRoom adaptations modifies the behavior of alerts in a room,



```
let obj = {
   field: new Signal(0),
   method: function () {
    return this.field.value = 1;
   }
};

obj.method();
```

```
let obj = {
   field: new Signal(0),
   method: function () {
    return this.field.value = 1;
   }
};
EMA.exhibit(obj, {field: "field"});
EMA.activate("field > 0");
obj.method();
EMA.layer({condition: "field > 0"});
```

```
let obj = {
   field: new Signal(0),
   method: function () {
    return this.field.value = 1;
   }
};
EMA.exhibit(obj, {field: "field"});
EMA.activate("field > 0");
EMA.layer({condition: "field > 0"});
obj.method();
```

Number of executions per n creations
n
1
a) Objects
b) Objects and layers
c) Objects and interfaces

Fig. 10. Three pieces of code used for the EMAjs performance evaluation: (a) Only creates objects, (b) creates objects and deploys layers, and (c) creates objects and interfaces.

whenever a user is in it. The BabyRoom adaptation has a similar effect, however, the behavior of both adaptations differs. As a consequence, using COP, if there is a user in a room, and a baby in other room of the house, the behavior observed upon an alert will correspond to the same behavior. This an undesired behavior, as we unintentionally capture a global behavior for all adaptations.

EMA makes it possible to scope adaptations for each specific situation, offering a greater flexibility to apply adaptations and avoid undesired/unintentional behavior. Using the layer activation for a specific object instance, as shown in Snippet 10, for the case of activating the specialized behavior for a baby in a room, this behavior will only be visible for that specific object instance. All other rooms will behave according to their corresponding occupancy.

Snippet 10: Scoping adaptations to specific object instance

```
bedroom.babyInRoom();
EMA.activate(atMostOne(BabyCondition), bedroom);
```

5.2. Performance evaluation

The paper proposal focuses on defining an expressive modular activation mechanism for COP; hence, we have not sacrificed any potentially valuable feature on the basis of its expected cost. Even so, it is valuable to have an outline of performance for programming language proposals. Therefore, we carried out a preliminary performance evaluation using EMAjs. We used Nodejs (v16) [21] on Macbook Pro (2020), 2 GHz Quad-Core Intel Core i5 with 16 GB of RAM running macOS Monterey, and EMAjs GitHub revision 903fdae (May 18, 2021).

Fig. 10 shows the three JavaScript programs executed to evaluate EMAjs performance. The first program creates up to 600 objects (Fig. 10a), the second one creates/deploys up to 600 objects/layers with one interface (Fig. 10b), and the last one creates up to 600 objects/interfaces with one layer (Fig. 10c). Using these three programs, Fig. 11 shows a preliminary performance evaluation of the current EMAjs implementation. The figure shows the average time in ms (y-axis) of 100,000 executions vs the creation of objects/layers/interfaces (x-axis), respectively for each of the programs. Although EMAjs does not observe every statement trying to activate a layer or execute a partial method (Figs. 10b and 10c), the current implementation is clearly slower than a program without EMAjs (Fig. 10a). Additionally, an incremental use of interfaces affects performance. These results could be due to the propagation of data stream in an unoptimized and naive reactive programming implementation [58]. In JavaScript, we can find proposals like Flapjax [59] and RxJS [60] that have also faced performance issues to extend this language with reactive programming. Note that in the two programs that use layers (Figs. 10b and 10c), we can see that the scope strategy is only executed only once. This is so because the scope strategy overhead is hidden compared to remaining statements executed.

5.3. Threats to validity

We now present the threats to the validity of our evaluation, and discuss possible limitations of our approach.

Threats to external validity refer to the generalization of the results from our evaluation. The evaluation presented in this work is based on the development of two applications coming from different domains, to account for different types of interactions between adaptations for different contexts. Moreover, the applications are of different sizes to remove possible bias with respect to the contexts used in the application, and the conditions in which they should take place. While many different types of interactions are covered by the applications showcased in the validation, we cannot claim that these are exhaustive. Therefore, EMA may not cover all possible interactions between adaptations. However, we argue that the flexibility of interfaces and the possibility to extend the activation language, can ease this problem by extending our proposal.

Threats to construct validity refer to the cases in which the evaluation is not exhaustive or selective enough. In this case, the modularity analysis of the smart home application is restricted to the implementations provided by the authors, experts in exploiting the modularity of the three application versions. A wider evaluation with users providing their own implementations of the three application versions can provide a more faithful evaluation of the modular capabilities of both COP and EMA, as it would be used in the wild.

The proposed evaluation is successful in validating the objectives and contributions of EMA. However, we note that further empirical evaluation could strengthen the arguments of the paper with the contributions of practitioners. Moreover, this would help us in positing EMA as a viable and usable technology with benefits for developers. Furthermore, the application of EMA to more domains can help us in positioning its contributions within the software development industry.

6. Related work

Different proposals related to ideas behind EMA exist. In [61], the author presents software development based on *open implementations*, which push software modularity through guidelines for the construction of abstractions that can encapsulate customized behavior semantics. In this line, proposals for JavaScript can be found [62,63]. Particularly in COP, for example, we can find ContextJS [31] as an open implementation for layer composition, which allows developers to use JavaScript to implement customized scope strategies. We next discuss proposals related to the expressive declaration of scope strategies and decoupling between layers and the base code.

Group behavior. Group behavior is proposed for COP [24,25] to enable the activation of a group of layers, according to the satisfaction of a predicate associated to partial behavior definitions. Here, partial methods are associated with a predicate expressing conditions about the internals of the system or its surrounding environment. Whenever a method is called, all predicates of all partial methods matching the method signature are evaluated. The behavior selected corresponds

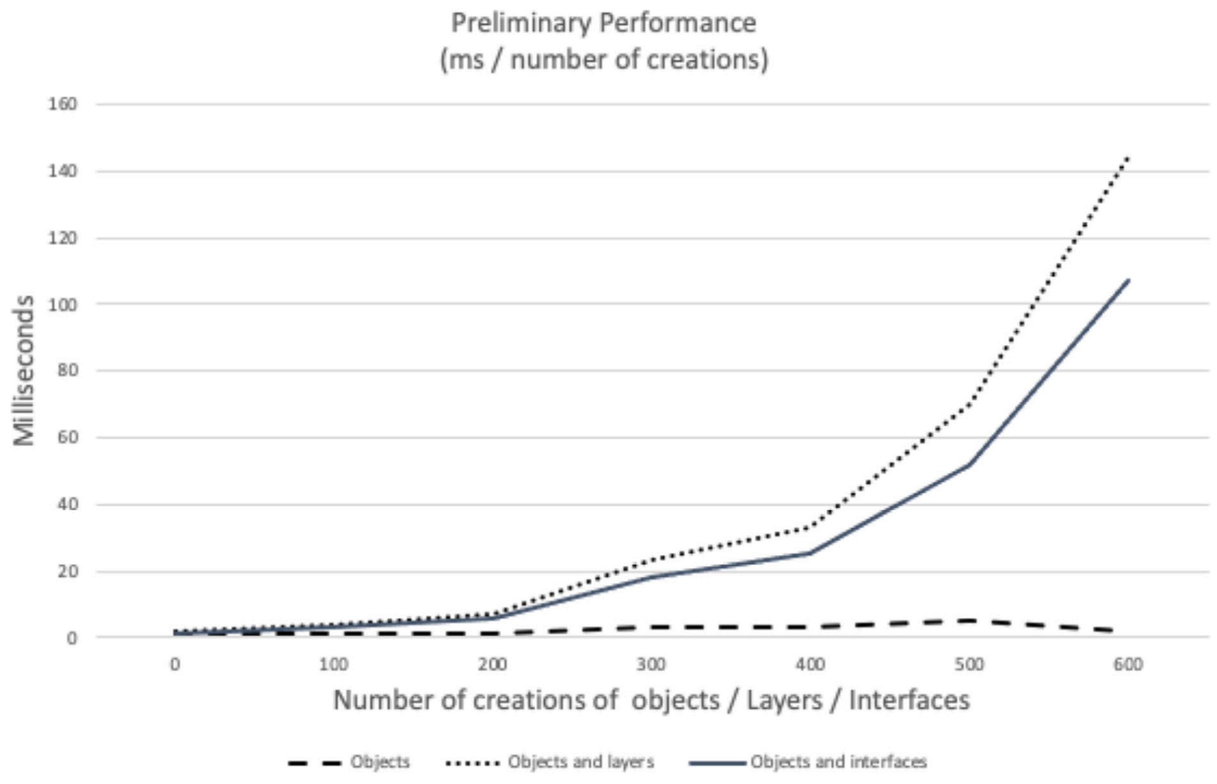


Fig. 11. A preliminary performance evaluation about EMAjs.

to valid predicates. The definition of group behavior is beneficial to adapt the behavior of different program entities beyond their class definition. Groups of objects can be defined explicitly by programmers or implicitly by a specific shared property [64]. Although EMA does not directly support the feature of group behavior, the expressive API for scoping allows developers to emulate this feature because of two reasons. First, the conditional to activate a layer might be shared among (several) objects, allowing to activate a set of layers (with specific behaviors) when a certain conditional is satisfied. Second, the group behavior activations can be accurate in different contexts using the expressive API for scoping available in EMA.

Decoupling layers and base code. This challenge is not completely new in the COP community. For example, the authors in [65] present a software architecture to modularly integrate each abstraction and mechanism of COP into the software development process. However, the architecture is particularly proposed for context-aware systems [1], where dynamic behavior adaptations is part of main functionalities of a system. Regarding the explicit decoupling between abstractions from different paradigms, we find different proposals [52,53,55] that decouple implicit dependencies between aspects of aspect-oriented programming [16] and base code. For instance, in Join Point Interfaces (JPI) [53], base code developers must *explicitly* establish what *join points* are exhibited by objects of a class; allowing the modular reasoning of the application of an aspect. Inspired by JPI, EMA uses interfaces to decouple references of variables used in conditionals used to activate layers and variables in base code. These interfaces additionally allow developers to exhibit expressions composed of variables or method executions that come from different objects. As a conclusion, to the best of our knowledge, EMA presents the first approach to *explicitly* decouple the layers and base code of an application in the COP community.

7. Conclusion and future work

Our lives are being saturated with many computing devices, and systems that dynamically adapt their behavior according to their surrounding execution environment (*e.g.,* users' preferences). Such systems are becoming ever bigger and more complex. COP [2] provides abstractions, like layers, to modularly implement these systems as a means to reduce their complexity. A layer is used to dynamically adapt method's behavior when the layer becomes active. In the literature, there is a significant number of activation mechanisms, which have been tailored to satisfy specific developers' needs [2,5–14]. Therefore, when developers face unforeseen needs, they end up tweaking layer declarations in contortive ways, or creating new specialized activation mechanisms. To address the previous issue, this paper proposes to complement two ideas [19,66] to present an Expressive and Modular Activation mechanism (EMA), which allows developers to define customized mechanism scope strategy and decouple the implicit dependency between activation conditionals and base code. As a consequence, layers improve their reuse, flexibility, and their modularity to be applied in different scenarios of adaptive software systems.

This paper shows the usability and effectiveness of EMA in developing expressive and modular context-aware systems. In the future, we strive to offer a more robust report on the benefits and disadvantages of EMA, by integrating EMA to current COP language implementations, exploring different base abstractions from those used in EMAjs, like ServalCJ [11,14] or Subjective-C [5].

CRediT authorship contribution statement

Paul Leger: Conceptualization, Methodology, Software, Validation, Investigation, Resources, Writing – original draft, Writing – review & editing, Project administration, Funding acquisition. **Nicolás Cardozo:** Software, Validation, Resources, Writing – original draft, Writing – review & editing. **Hidehiko Masuhara:** Conceptualization, Methodology, Investigation, Visualization, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] M. Satyanarayanan, Pervasive computing: Vision and challenges, IEEE Pers. Commun. 8 (4) (2001) 10–17.
[2] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, J. Object Technol. 7 (3) (2008) 125–151.
[3] A. Dey, G. Abowd, Towards a better understanding of context and context-awareness, in: Proceedings of the Workshop on the What, Who, Where, When, Why and How of Context-Awareness, Karlsruhe, Germany, 2000.
[4] P. Costanza, R. Hirschfeld, Language constructs for context-oriented programming: An overview of ContextL, in: Proceedings of the Dynamic Languages Symposium, San Diego, California, 2005, pp. 1–10.
[5] 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 Third International Conference on Software Language Engineering, Eindhoven, The Netherlands, 2011, pp. 246–265.
[6] S. González, K. Mens, M. Colacioiu, W. Cazzola, Context traits: Dynamic behaviour adaptation through run-time trait recomposition, in: Proceedings of International Conference on Aspect-Oriented Software Development, Fukuoka, Japan, 2013, pp. 209–220.
[7] N. Cardozo, S. González, K. Mens, Uniting global and local context behavior with context Petri nets, in: Proceedings of the International Workshop on Context-Oriented Programming, Beijing, China, 2012, pp. 1–3.
[8] S. Ramson, J. Lincke, R. Hirschfeld, The declarative nature of implicit layer activation, in: Proceedings of the International Workshop on Context-Oriented Programming, COP, London, UK, 2017, pp. 7–16.
[9] 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, Porto de Galinhas, Brazil, 2011, pp. 253–264.
[10] M. von Löwis, M. Denker, O. Nierstrasz, Context-oriented programming: Beyond layers, in: Proceedings of International Conference on Dynamic Languages, Lugano, Switzerland, 2007, pp. 143–156.
[11] T. Kamina, T. Aotani, H. Masuhara, Push-based reactive layer activation in context-oriented programming, in: Proceedings of the 9th International Workshop on Context-Oriented Programming, COP, Barcelona, Spain, 2017, pp. 17–22.
[12] T. Watanabe, A simple context-oriented programming extension to an FRP language for small-scale embedded systems, in: Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition, Amsterdam, Netherlands, 2018, pp. 23–30.
[13] T. Aotani, T. Kamina, H. Masuhara, Unifying multiple layer activation mechanisms using one event sequence, in: Proceedings of 6th International Workshop on Context-Oriented Programming, COP, Uppsala, Sweden, 2014, pp. 1–6.
[14] T. Kamina, T. Aotani, H. Masuhara, Generalized layer activation mechanism for context-oriented programming, LNCS Trans. Modul. Compos. 9800 (2016) 123–166.
[15] K. Mens, B. Duhoux, N. Cardozo, Managing the context interaction problem: A classification of conflict resolution techniques in dynamically adaptive software systems, in: Proceedings of the International Workshop on Live Adaptation of Software Systems, Brussels, Belgium, 2017, pp. 8:1–8:6.
[16] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, A. Mendhekar, Aspect-oriented programming, in: M.M. (general editor), et al. (Eds.), Special Issues in Object-Oriented Programming, 1996.
[17] K. Gybels, J. Brichau, Arranging language features for more robust pattern-based crosscuts, in: Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development, AOSD, Boston, MA, USA, 2003, pp. 60–69.
[18] N. Cardozo, A declarative language for context activation, in: Proceedings of the International Workshop on Context-Oriented Programming, Amsterdam, The Netherlands, 2018.
[19] P. Leger, H. Masuhara, I. Figueroa, Interfaces for modular reasoning in context-oriented programming, in: International Workshop on Context-Oriented Programming and Advanced Modularity, Virtual Event, USA, 2020, pp. 1–7.
[20] EMAjs Website, A JavaScript library of an expressive and modular activation mechanism for context-oriented programming, 2022, https://github.com/pragmaticslaboratory/EMAjs. Last visited: 11/08/2022.
[21] NodeJS, A JavaScript runtime built for the server side, 2021, URL https://nodejs.org, (v16.0).
[22] Google Chrome, A free and open-source web browser, 2022, URL https://www.google.com/chrome, (v104.0).
[23] Firefox, A free and open-source web browser, 2021, URL https://www.mozilla.org, (v103.0).
[24] 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, no. 6144, Malaga, Spain, 2010, pp. 66–81.
[25] E. Bainomugisha, J. Vallejos, C. De Roover, A. Lombide Carreton, W. De Meuter, Interruptible context-dependent executions: A fresh look at programming context-aware applications, in: Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software Proceedings, Tucson, Arizona - USA, 2012.
[26] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, K. Kawauchi, Event-specific software composition in context-oriented programming, in: Software Composition, Malaga, Spain, 2010, pp. 50–65.
[27] É. Tanter, Beyond static and dynamic scope, in: Proceedings of the 5th ACM Dynamic Languages Symposium, DLS, Orlando, FL, USA, 2009, pp. 3–14.
[28] É. Tanter, Expressive scoping of dynamically-deployed aspects, in: Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development, AOSD, Brussels, Belgium, 2008, pp. 168–179.
[29] É. Tanter, J. Fabry, R. Douence, J. Noyé, M. Südholt, Expressive scoping of distributed aspects, in: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, AOSD, Charlottesville, USA, 2009, pp. 27–38.
[30] R. Toledo, P. Leger, É. Tanter, AspectScript: Expressive aspects for the web, in: Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development, AOSD, Rennes and Saint Malo, France, 2010, pp. 13–24.
[31] J. Lincke, M. Appeltauer, B. Steinert, R. Hirschfeld, An open implementation for context-oriented layer composition in contextjs, Sci. Comput. Programm. 76 (12) (2011) 1194–1209.
[32] R. Hirschfeld, P. Costanza, M. Haupt, An introduction to context-oriented programming with ContextS, in: Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers, Springer, Braga, Portugal, 2008, pp. 396–407.
[33] M. von Löwis, M. Denker, O. Nierstrasz, Context-oriented programming: Beyond layers, in: Proceedings of the International Conference on Dynamic Languages, Lugano, Switzerland, 2007, pp. 143–156.
[34] 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, Montreal, Quebec, Canada, 2007, pp. 77–88.
[35] T. Kamina, T. Tamai, Towards safe and flexible object adaptation, in: International Workshop on Context-Oriented Programming, ACM, Genova, Italy, 2009, pp. 1–6.
[36] G. Salvaneschi, C. Ghezzi, M. Pradella, ContextErlang: A language for distributed context-aware self-adaptive applications, Sci. Comput. Program. 102 (2015) 20–43.
[37] R. Hirschfeld, M. Perscheid, C. Schubert, M. Appeltauer, Dynamic contract layers, in: 25th Symposium on Applied Computing, ACM, Lausanne, Switzerland, 2010, pp. 2169–2175.
[38] 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, Maribor, Slovenia, 2010.
[39] T. Kamina, T. Aotani, H. Masuhara, A unified context activation mechanism, in: Proceedings of the 5th International Workshop on Context-Oriented Programming, Montpellier, France, 2013, pp. 1–6.
[40] R. Hirschfeld, H. Masuhara, A. Igarashi, L: Context-oriented programming with only layers, in: Proceedings of the 5th International Workshop on Context-Oriented Programming, ACM, Montpellier, France, 2013, pp. 1–5.
[41] B. Maingret, F. Le Mouël, J. Ponge, N. Stouls, J. Cao, Y. Loiseau, Towards a decoupled context-oriented programming language for the Internet of Things, in: Proceedings of the 7th International Workshop on Context-Oriented Programming, Prague, Czech Republic, 2015, pp. 1–7.
[42] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon, Common lisp object system specification, SIGPLAN Not. 23 (1988) 1–142.
[43] S. González, K. Mens, A. Cádiz, Context-oriented programming with the ambient object system, J. UCS. 14 (20) (2008) 3307–3332.
[44] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: Proceedings of the European Conference on Object-Oriented Programming, Vol. 2072, 2001, pp. 327–354.
[45] L.C. Kats, E. Visser, The spoofax language workbench: Rules for declarative specification of languages and IDEs, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, Reno/Tahoe, USA, 2010, pp. 444–463.
[46] J. Ponge, F. Le Mouël, N. Stouls, Golo, a dynamic, light and efficient language for post-invokedynamic JVM, in: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, 2013, pp. 153–158.
[47] K. Sawada, T. Watanabe, Emfrp: A functional reactive programming language for small-scale embedded systems, in: International Conference on Modularity, 2016, pp. 36–44.

[48] D. Ungar, H. Ossher, D. Kimelman, Korz: Simple, symmetric, subjective, context-oriented programming, in: Onward! 2014 - Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Part of SPLASH 2014, Portland, USA, 2014, pp. 113–131.

[49] A. Elyasaf, Context-oriented behavioral programming, Inf. Softw. Technol. 133 (2021) 106504.

[50] S. González, Programming in Ambience: Gearing Up for Dynamic Adaptation to Context (Ph.D. thesis), Université catholique de Louvain, 2008, Coll. EPL 211/2008. Promoted by Prof. Kim Mens.

[51] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, U.A. mann, A metamodel family for role-based modeling and programming languages, in: International Conference on Software Language Engineering, Västerås, Sweden, 2014, pp. 141–160.

[52] J. Aldrich, Open modules: Modular reasoning about advice, in: European Conference on Object-Oriented Programming, Glasgow, UK, 2005, pp. 144–168.

[53] E. Bodden, É. Tanter, M. Inostroza, Join point interfaces for safe and flexible decoupling of aspects, 23 (1), 2014, pp. 1–41.

[54] W.G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, H. Rajan, Modular software design with crosscutting interfaces, 23 (1), 2006, pp. 51–60.

[55] F. Steimann, T. Pawlitzki, S. Apel, C. Kästner, Types and modularity for implicit invocation with implicit announcement, ACM Trans. Softw. Eng. Methodol. 20 (1) (2010) 1–43.

[56] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, M. Perscheid, A comparison of context-oriented programming languages, in: International Workshop on Context-Oriented Programming, Genova, Italy, 2009, pp. 1–6.

[57] N. Cardozo, K. Mens, Programming language implementations for context-oriented self-adaptive systems, Inf. Softw. Technol. 143 (2022) 106789.

[58] E. Amsden, A Survey of Functional Reactive Programming Concepts, Implementations, Optimizations, and Applications, Tech. Rep., Rochester Institute of Technology, 2011.

[59] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi, Flapjax: A programming language for Ajax applications, in: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, Orlando, Florida, USA, 2009, pp. 1–20.

[60] Apache, RxJS: A reactive extensions for JavaScript (v7.5.6), 2018, URL https://rxjs.dev.

[61] G. Kiczales, Towards a new model of abstraction in software engineering, in: Proceedings of the IMSA 92 Workshop on Reflection and Metalevel Architectures, Tokyo, Japan, 1992.

[62] P. Leger, É. Tanter, R. Douence, Modular and flexible causality control on the web, Sci. Comput. Program. 78 (9) (2013) 1538–1558.

[63] P. Leger, É. Tanter, H. Fukuda, An expressive stateful aspect language, Sci. Comput. Program. 102 (2015) 108–141.

[64] P. Rein, S. Ramson, J. Lincke, T. Felgentreff, R. Hirschfeld, Group-based behavior adaptation mechanisms in object-oriented systems, IEEE Softw. 34 (6) (2017) 78–82.

[65] K. Mens, N. Cardozo, B. Duhoux, A context-oriented software architecture, in: Proceedings of the 8th International Workshop on Context-Oriented Programming, COP, Rome, Italy, 2016, pp. 7–12.

[66] N. Cardozo, Emergent software services, in: Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Amsterdam, Netherlands, 2016, pp. 15–28.