Programming Language Implementations for Context-Oriented Self-Adaptive Systems

Nicolás Cardozo*

Systems and Computing Engineering Department, Universidad de los Andes. Bogotá, Colombia.

Kim Mens

ICTEAM, Université catholique de Louvain. Louvain-la-Neuve, Belgium.

Abstract

Context: The context-oriented programming paradigm is designed to enable self-adaptation, or dynamic behaviour modification of software systems, in response to changes in their surrounding environment. Context-oriented programming offers an adaptation model, from a programming language perspective, that maintains a clean modularisation between the application and adaptation logic, as well as between the components providing adaptations. **Objective:** We use three implementation techniques for context-oriented programming languages to assess their appropriateness to foster self-adaptive systems. These approaches take advantage of the capabilities offered by the host programming language to realise self-adaptation as proposed by context-oriented languages. **Method:** We evaluate each of these approaches by assessing their modularity and complexity when defining adaptations, and by comparing their run-time performance on a simple benchmark. **Results:** Our results show a higher modularity than that for common architecture-based self-adaptive systems, while maintaining comparable performance. **Conclusion:** We conclude that context-oriented programming is an appropriate paradigm to realise self-adaptation.

Keywords: Context-oriented programming, Self-adaptive systems, Language engineering, Programming language abstractions.

1. Introduction

Modern software systems are no longer developed under the assumption that all system entities and boundaries are completely known and defined beforehand. Rather, the development of software systems is shifting towards systems highly aware of their surrounding environment, which is in continuous change. Such systems, called self-adaptive, context-aware, or context-oriented systems, are able to dynamically adapt their behaviour with respect to internal (*self*) and external (*context*) conditions of the execution environment [52].

There are two main approaches to realize context-oriented or self-adaptive systems. Systems can be defined to use *external* or *internal* adaptation engines. The main difference resides in where the adaptation logic of the system is implemented [20]. External adaptation engines claim to provide a more modular structure of the system, where the adaptation logic resides in external modules independent from those of the application logic. Internal adaptation engines, in contrast, typically implement both the adaptation and application logic inside the system itself.

Since external adaptation engines expose the adaptation modules used for a particular system, these modules can be customised and reused to manage adaptation of other related systems with ease. Such adaptation engines are thus regarded as highly modular and reusable. Nonetheless, to be able to reuse or customize these adaptations,

*Corresponding author. Universidad de los Andes, Systems and Computing Engineering Department, FLAG lab. Bogotá, Colombia. e-mail: n.cardozo@uniandes.edu.co. Phone: +57 1 339 4949

Email addresses: n.cardozo@uniandes.edu.co (Nicolás Cardozo), kim.mens@uclouvain.be (Kim Mens)

external adaptation engines need to abstract their reusable components to achieve sufficient genericity, which raises their architectural footprint.

Internal adaptation engines are mostly regarded by the self-adaptive community as a less flexible solution for adaptivity, because they are considered to be tightly coupled with the system's implementation. If the adaptation and application logic are too interwoven, this indeed gives rise to maintainability, reusability, and evolution problems. Therefore, the use of external adaptation engines is often promoted over internal adaptation engines for the development of self-adaptive systems [52].

In this paper we argue that it is possible to design internal adaptation engines that cleanly separate the adaptation logic from the application logic. Such internal adaptation engines can provide a viable alternative for the implementation of self-adaptive systems, exhibiting even more flexibility than their external counterparts, precisely because they do not suffer from added architectural complexity.

One proposal for the development of internal adaptation engines for self-adaptive systems is that offered by Context-oriented Programming (COP) [13, 30]. Such adaptation engines avoid architectural clutter by pushing it down to the programming language design, and provide appropriate language abstractions for dealing with both the adaptation and application logic independently.

This paper showcases different implementation techniques for COP languages and illustrates how they achieve self- and context-adaptation[1] at the programming language level. Our claim is that COP provides a clear independence between adaptation and application logic, as well as a transparent adaptation logic and management interface to developers.

We explore three different COP implementation techniques to enable dynamic adaptation of software systems with respect to their surrounding execution environment. The techniques discussed are representative of a range of techniques used in existing COP approaches to enable adaptation at the programming language level. In particular, the implementation techniques showcased hereinafter are inspired by our work in COP. Nonetheless, we note that the specific languages showcased cover the variability of the four key dimensions to achieve adaptivity in COP languages [54]: (1) the host language's dynamic adaptation and composition capabilities, (2) the adaptation selection mechanisms, (3) the scope of adaptations, and (4) the separation of adaptation and application logic.

COP languages provide a language-native approach to develop modular software systems that can adapt dynamically to a variety of contexts. The dedicated programming language abstractions provided by COP languages make it possible to develop highly flexible self-adaptive systems in which the adaptation and application logic are cleanly modularised [54].

We focus our analysis on the different implementation techniques to achieve dynamic adaptation for COP languages, making them suited to build self-adaptive software systems. The desired properties of self-adaptive systems we expect internal adaptation engines to satisfy are taken from the literature on taxonomies of self-adaptive systems [52, 31], and existing properties and taxonomies of context-aware systems [30, 57, 46]. In summary, we want internal adaptation engines that are: **(1) smart**, by choosing the most appropriate system behaviour with respect to the current execution context; **(2) open** to interact with and adapt to different (new) agents that operate in the system's surrounding environment; **(3) resilient** to changes, by offering the possibility to implement graceful degradations of the systems' behaviour to particular contexts; **(4) extensible**, since unforeseen adaptations can be added on-the-fly, simply by adding new behavioural variations together with a high-level specification of the contexts to which they apply; **(5) highly available**, as the system does not need to be stopped to introduce new behaviour or bug fixes [8]; **(6) customizable** to many different variations made available for particular situations; **(7) modular**, as (groups of) adaptations can be bundled separately from each other and the application logic; **(8) fine-grained**, allowing adaptations of existing behaviour at the level of granularity of individual methods; **(9) concise**, as adaptations consist only of the fine-grained adapted behaviour they define and their definition is decoupled from unrelated existing adaptations and application logic; and **(10) understandable**, since the application logic is not cluttered by adaptation management concerns, given that this mechanism is internalized in the language.

This paper presents and analyses three imple-

[1] From this point onwards, given our focus on context-oriented programming languages, we use *adaptation*, to mean both *self- and context-adaptation*.

mentation techniques to realize internal adaptation engines at a programming language level: *context-dependent method dispatch*, *method replacement*, and *fine-grained composition*. Each technique pertains to a specific point in the system's execution to realise the adaptation. The first technique extends the language's method dispatching mechanism to dynamically select the appropriate behaviour to execute based on the context. The second technique uses method implementation exchange to dynamically change the pointer of method implementations as contexts change. The third technique enables the dynamic composition of fine-grained software components (*i.e.,* methods) to include or withdraw behaviour from the system according to the context.

The reason to explore these implementation techniques is twofold. First, we have explored these techniques in our previous research on COP since its inception. Second, across COP languages, we observe that the base of the techniques used in other implementations falls back to one of the three techniques we explore in the paper. However, other techniques are available, which we discuss further in Section 5.

For each of the implementation techniques we analyze their appropriateness from three perspectives: expressiveness, modularity, and performance. *Expressiveness* is evaluated by comparing the differences between the implementations to express a common case study of a smart city guide application. *Modularity* is evaluated empirically by observing the independence of the implemented components. *Performance* is evaluated through dedicated mini benchmarks measuring the different adaptation techniques against *de facto* techniques to enable adaptations in software systems. Using our evaluation, we conclude that internal adaptation engines provided by COP languages successfully achieve maintainability, reusability, and high adaptivity, at an acceptable performance cost, and are therefore a viable option to foster self-adaptive systems.

The remainder of this paper is organized as follows. Section 2 presents an overview of COP implementations and the main language abstractions used in COP to realise self-adaptation. That section also presents our running example of a smart city guide, as implemented in the Subjective-C COP language. Section 3 details the explored implementation techniques to enable adaptation at the programming language level. Each of these implementations is associated to a particular COP language. Section 4 presents an evaluation of each technique with respect to its expressiveness, modularity, and performance. Section 5 discusses related approaches existing in the COP community and related language-level techniques to enable dynamic software adaptation. Finally, Section 6 draws lessons from our experience in developing self-adaptive systems with internal adaptation engines, and offers perspectives of future work.

2. Context-oriented Programming

Over the last decade, there has been increasing interest of the programming language community to provide language-level abstractions and support to realise and manage dynamic behavioural variations. The objective of this body of research is to provide dedicated programming facilities to foster software systems in the vision of Ubiquitous Computing [65] and Ambient Intelligence [58]. The COP paradigm is proposed to cater precisely to this necessity [13, 30].

COP provides support for developing software systems with specific behaviour variations according to particular situations occurring in the system's surrounding environment. Such behaviour is made available whenever the respective situations are sensed by the system, for example, through a sensor network. Situations for which the system can exhibit specialised behaviour can be external (*e.g.,* weather conditions, time of day), internal system events (*e.g.,* system preferences, CPU load) or user preferences (*e.g.,* language, age group). In COP, no explicit distinction between these kinds of situations is made and they are commonly referred to as *contexts*. Henceforth, we use context to mean both external or internal events, and user preferences.

2.1. COP Language Abstractions

Different programming languages (or extensions of existing programming languages) have been developed to realise the COP paradigm [54, 2]. COP languages allow developing applications that feature behavioural variations that are seamlessly composed with, or withdrawn from, the main systems' application logic, responding to the current execution context. We now define the main COP concepts and terminology used throughout the rest of this paper. This terminology covers the main building blocks to enable dynamic behavioural variations that all COP languages provide (in some

form or another) regardless of their specific implementation.

Context: Corresponds to a specific situation of the system's surrounding environment that is semantically meaningful for the system. Contexts[2] are defined as independent (first-class) system entities.

Behavioural variation: Defines a specialised behaviour that the system should exhibit under a particular situation. This can be defined as an independent module or feature of the system or within the system modules (*e.g.,* classes) it adapts.

Adaptation: Refers to a specific context and all of its associated behavioural variations.

Context activation/deactivation: Happens as a response to changes sensed in the system's surrounding environment. Context activation and deactivation, make available or unavailable all behavioural variations associated with a context. This is done by allowing the dynamic composition or withdrawal of behavioural variations with or from the system's application logic.

The abstractions introduced by COP languages for the definition of contexts and behavioural variations, allow programmers to reduce the tangling and scattering of adaptation logic with the application logic, as opposed to claims about internal adaptation engines [20, 52]. Moreover, the adaptation logic is transparently reusable because adaptation composition is not specific to a particular application or set of adaptations, but rather all applications compose adaptations in the same fashion. Furthermore, the declaration of contexts and their associated behavioural variations could be reused for different systems, assuming they are in accordance with the purpose of those other systems [8]. Finally, we claim that using COP to build self-adaptive software systems can prove beneficial for system evolution (given that upgrades to the application logic can be introduced dynamically as new behavioural variations), and maintainability (as both adaptation and application logic are cleanly modularised) [47].

2.2. Case Study

As a case study of a dynamically adaptive application, we consider a *smart city guide* [38, 11], a real-world mobile application that enables tourists visiting a city to navigate through its Points of Interest (POIs). The application provides the possibility to create and customise city tours and navigate through them based on a selection of POIs. These features can be enhanced further according to the surrounding environment or user preferences.

The smart city guide application constitutes a large case study for context-aware self-adaptive systems. Figure 1 depicts the behavioural variations and the contexts used in the definition of such an application, represented in terms of feature diagrams, following a feature-based context-oriented model [17]. These features are elaborated upon in the following functional requirements:

Tour Creation & Selection: Users can create and select city tours based on a list of available POIs. A tour can be followed in two modes: if the device has a `GPSAntenna`, the application offers a `GuidedTour` mode which guides users throughout the city according to predefined `Itineraries` of POIs. The `TimeOfDay` can affect the order in which these POIs are visited, to take into account their visiting hours, or to display different images depending on the time of day when the application is used. For example, during the `Day` only daytime images of buildings and statues are shown, so that they can be recognised easily by users. In `FreeTour` mode, users can walk freely around the city and receive directions to nearby POIs. Similarly to the previous case, direction to, and information about POIs can be tailored to `Day` or `Night` time.

City Navigation: Provides the possibility to navigate between POIs using either detailed `Directions` or by following a `Map` using a `Compass`. For example, in `FreeTour` mode, where no `GPSAntenna` is available, the `Compass` will be used. Upon arrival at a POI, a `POIDescription` containing its information is displayed to the user. If a `Wifi` connection is available, the application can allow users to fetch additional multimedia information about the POI.

POI Display & Information: A POI's description can be customised according to various `UserPreferences` (*e.g.,* `UserLanguage` or `Age`). The `UserLanguage` context enables the application to adapt its displayed information to a particular language, such as `English` or `French`. The language to use is determined either by the geographical position of the user (*e.g.,* UK or France), or is preselected by

[2] In the COP literature, the technical term *layer* is also used to refer to contexts.

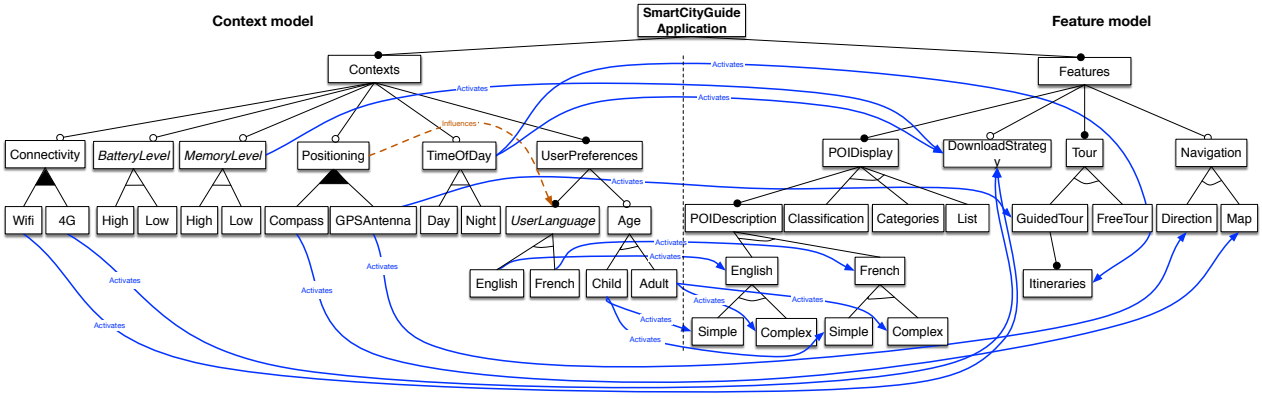


Figure 1: Excerpt of the context and feature model of the *smart city guide* application

the user in the application preferences. We restrict the application to offer only one language at a time.

The left-hand side of Figure 1 shows the principal contexts declared for the application, based on the usage situations in which specialised behaviour is required. There are 22 contexts specialising the behaviour of the four main features. For example, the functionality for `POIDisplay` is specialised to use audiovisual descriptions for POIs, whenever there is `Wifi` connectivity; or to adapt the complexity of the displayed information to the `Age` category of the user, *e.g.,* `Child` or `Adult`.

3. Extending Programming Languages to Support Self-Adaptation

COP provides language abstractions that enable the development of self-adaptive systems, in which the adaptation engine is internal to the system. This means that the logic to adapt a program at run time is intertwined with the programming language, and is accessible to applications by means of a dedicated API.

We have explored different programming language approaches to enable dynamic adaptation of software systems' behaviour with respect to the surrounding environment [23, 25, 49, 26]. In the following, we present the underlying development techniques we have explored to build COP languages.

3.1. Context Definition

To enable adaptations, we first need to define *contexts*. Contexts are first-class entities of the language, defined (through syntactic sugar) as objects in the underlying host language. For example, in Subjective-C (a COP language implemented on top of Objective-C), contexts are defined using the `@context(Wifi)` abstraction, yielding a new context object named `Wifi`, as shown in Snippet 1.

```
SCContext* wifi = [[SCContext alloc]
    initWithName:@"wifi"];
```

Snippet 1: Definition of context objects in Subjective-C

This definition is merely syntactic sugar for the context definition above, where `SCContext` is a regular Objective-C object. Each context object defines a set of all possible `behavioralVariations` associated with that context. Every time a (new) behavioural variation is associated with this context, it is added to this set.

Dynamic behavioural variations can be characterised by their run-time *composition*, *selection* and *scoping* mechanisms.

3.2. Dynamic Behaviour Composition

Dynamic behaviour composition is the ability of COP languages to dynamically introduce or withdraw behavioural variations. That is, whenever a context is sensed in the surrounding environment, the system must know which behavioural variations are associated to that context. Whenever a context becomes active, its associated behavioural variations get added to the system; whenever the context is no longer present in the surrounding environment, its associated behavioural variations are withdrawn from the system.

We have explored three main techniques to enable the dynamic composition of behavioural variations in different COP languages: (1) *context-dependent method dispatch*, in Ambience [23],

(2) *method replacement*, in Subjective-C [25], and (3) *fine-grained composition*, in Context Traits [26]. These techniques explore different implementation strategies, taking advantage of particular features provided by the host programming language. Further techniques to define adaptations at the language level are discussed in Section 5.

3.2.1. Context-dependent method dispatch

When adapting the system's behaviour to its surrounding execution environment, the system should choose the behavioural variation that is most appropriate to the current context. This choice can be made during method dispatch, by appending the context information to every method call.

Multiple dispatch has been explored as a particular implementation technique to enable dynamic adaptation in COP languages (notably in ContextL [13] and Ambience [24]). Specifically, we focus on the implementation of Ambience, a Lisp-like COP language. With *single dispatch*, methods are looked up dynamically, based on the type of the receiver of a message. *Multiple dispatch* mechanisms can use multiple message arguments to choose the concrete method implementation to call. Consider for example the implementation of our smart city guide using Common Lisp, as shown in Snippet 2, where calls to `display-poi-info` are dispatched to the most appropriate method definition depending on the actual types of its three arguments `user`, `conn`, and `day-time`.

```
(defmethod display-poi-info ((user Adult) (conn
    Wifi) (day-time Night))
  ;;full night-time info with elaborate content
  ;;for adult users
  )
(defmethod display-poi-info ((user Child) (conn
    Wifi) (day-time Night))
  ;;full night-time info
  ;;with simplified content for children
  )
(defmethod display-poi-info ((user Child) (conn
    Edge) (day-time Day))
  ;;brief day-time info due to lower bandwidth
  ;;with simplified content for children
  )
```

Snippet 2: Example of multiple dispatch

From our experience in developing COP languages and the expected behaviour of context-aware systems, we have observed three properties that ease building dynamic adaptations using a context-dependent method dispatch technique: (1) The behaviour most appropriate to the current situation should be chosen based on the surrounding execution context [15]. (2) When looking up which method implementation to use, priority should be given to the context of execution over other arguments (to ensure the system behaviour is driven by the context). (3) Relations between context objects are needed to reduce the amount of arguments that need to be passed as arguments to a message.

A natural way of using multiple dispatch to enable dynamic adaptation, is to introduce context objects as additional method arguments, and dispatch to the most appropriate method taking into account all arguments. Behavioural variations for a given context will be selected only when the context argument in the message call matches that particular context. This approach requires the implicit and dynamic modification of all message calls, to introduce contexts as additional hidden arguments.

Messages are intercepted at the meta-level in Ambience's `amos::send` method, shown in Snippet 3. This method modifies a message to include contexts as its first argument, by in-lining the new argument directly in the method object, using the `prepend-slot` function (Lines 4 and 8-11).

```
1  ;;Message interception
2  (defmethod amos::send (selector arguments)
3    (declare (inline send-message))
4    (prepend-slot arguments (invoker-activation))
5    (send-message selector arguments))

7  ;;Add context object as the first parameter
8  (defun prepend-slot (object value)
9    (declare (inline add-slot))
10   (add-slot object :value value :index 0)
11   object)
```

Snippet 3: Metalevel insertion of contexts as message arguments for dynamic dispatch

Different situations or contexts can be sensed simultaneously in the surrounding execution environment, all of which should be taken into account in messages' context argument(s). To cater for all possible contexts, it would be necessary to define behavioural variations with different argument specifications for all possible combinations of contexts, which does not scale.

Instead, Ambience introduces the notion of the *current context* as a structured graph containing the information of all known contexts. In particu-

lar, this structure provides a representation of all active contexts in the system. This current context graph is added as first argument to each method call (Line 10 in Snippet 3).

Figure 2 shows an example of the *current context* graph for the smart city guide application with three active contexts: `@wifi`, `@night` and `@adult`. In the figure, all active contexts are represented in black. Other contexts defined in the system, but not currently active are shown in gray. To the right of the graph we have the contexts defined by users to represent situations in which specialized, *i.e.,* adaptive, behaviour should be present. To the left we see context objects internally generated and managed by the system, for example, to represent the composition of active contexts. The current context graph is built using delegation associations (the edges between the nodes in Figure 2) between the defined contexts in two ways: (1) composed contexts implicitly delegate to each of their constituents, and (2) explicit delegation links can be defined between context objects, according to the abstraction level in which they are defined. For example, a `@connectivity` context abstracts the `@wifi` context, so the latter context delegates to the former using the `(add-delegation @wifi @connectivity)` language construct. Following the context delegation graph from active user-defined contexts up the delegation associations, yields the current context (left-most node in Figure 2), as the composition of all active contexts. The particular context graph in Figure 2 is used to provide adaptations to the `display-poi-info` method.

The method lookup mechanism finds the correct method implementation for a given method call according to its arguments. In Ambience's multiple dispatch model, method lookup is based on *rank vectors* [55]. Let us use the `display-poi-info` method definitions from Snippet 2 to explain the way rank-vectors work. Suppose we have the following objects for each of the prototypes: `nicolas` an `Adult`, `home-wifi` a `Wifi` connection, and `evening` a `Night` time of day. The rank vectors are calculated according to the distance, in the delegation graph, between the concrete objects to the prototypes for the method definitions in Snippet 2. As such, for the first method definition in Snippet 2 we get a rank vector $(1, 1, 1)$ given that the distance in the delegation graph from the concrete object `nicolas` to its prototype `Adult` is 1 (all objects delegate directly to their prototype); similarly the distances from `home-wifi` to `Wifi`, and from `evening` to `Night` are 1 too, as each of these concrete objects delegate directly to the defined prototypes. For the second and third method definitions, the rank vectors are $(2, 1, 1)$, and $(2, 2, 2)$, respectively. Here, the concrete objects do not delegate directly to the prototype used in the method definition, requiring to find the closest common prototype (*i.e.,* the *meet*) instead. For example, `nicolas` is an `Adult` but the prototype mentioned in the method definition is a `Child` so there is no direct match. But both `Adult` and `Child` delegate to a common prototype `User`, so the distance for `nicolas` (to the closest *meet* `User` is 2). Using a similar reasoning we can find that the distance for `home-wifi` to `Edge` is 2 (distance to the closest *meet* `Connectivity`), and that the distance for `evening` to `Day` is 2 (closest *meet* is `TimeOfDay`). When looking for a method implementation, the selected method is the one with the smallest distance, in lexicographical order, to the message call. In our example, that would be the first implementation of `display-poi-info`, since $(1, 1, 1) < (2, 1, 1) < (2, 2, 2)$. Given that the current context graph is the first (implicit) argument in all methods, this method lookup mechanism ensures that methods are always chosen according to the most appropriate context.

Host languages providing capabilities to capture and dispatch method calls at run-time (*e.g.,* Objective-C, Common Lisp, Java[3]) are appropriate to implement self-adaptation via context-dependent method dispatch techniques. Context-dependent method dispatch mechanisms have been used too for the implementation of other COP languages. ContextL [13] uses a multiple dispatch mechanism to enable dynamic adaptations, while predicate dispatch is used in Lambic [62] and Flute [6]. More recently, dynamic invocation of methods has been used to implement the Congolo language [45].

3.2.2. Method replacement

An alternative technique to enable dynamic composition of behavioural variations is that of *method replacement*. The method replacement technique *reflectively* identifies all behavioural variations for a particular method at run time, whenever a context change is detected, and then *replaces* the current method behaviour by a more specialised one.

In order to implement the method replacement technique, it is necessary to access and modify a

[3] through `InvokeDynamic` or AOP method interception.

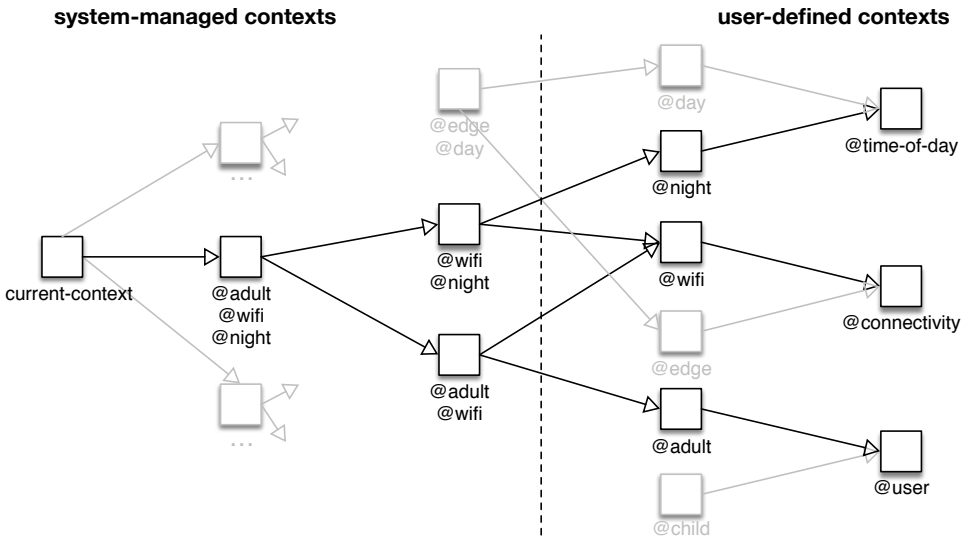


Figure 2: Current context graph in Ambience.

software system's information at run time. This can be done by taking advantage of the host language's metaobject protocol (MOP). A full MOP [60, 44, 39] enables *introspection* to inspect the structure of the system and the internal state of its objects. It enables *self-modification* if it is possible to modify the structure of the system. It enables *invocation* if is possible to call base-level operations (*e.g.,* method calls, or field assignment) reflectively. Host programming languages whose MOP offer at least introspection and self-modification (*e.g.,* Objective-C, CLOS, Smalltalk) are amenable to implement a method replacement technique for self-adaptation. As an example of this technique, we use our COP extension to Objective-C, named Subjective-C [25].

Objective-C objects have a variable called `isa` of type `Class` providing access to the class that the object is an instance of. This class can then be manipulated reflectively, for example to know if a given object understands a specific method, we can invoke the metalevel method `respondsToSelector:` on the object's class. Such reflective methods are provided by the metaclass `Class`, part of Objective-C's MOP. Self-modification of Objective-C applications can also be achieved reflectively through *method swizzling*, a technique offered by Objective-C's run-time to exchange the implementation of a given selector, by changing how selectors are mapped to underlying functions in a class' dispatch table.[4]

In the case of Subjective-C, we use method swizzling as a technique to replace method implementations when contexts get activated or deactivated. Method replacement in Subjective-C consists of two stages, as shown by the `updateDispatch` method in Snippet 4 (Line 14). First, we identify all behavioural variations for a particular method of the application. All methods in a Subjective-C application are defined as Subjective-C methods, `SCMethod`. Each `SCMethod` consists of its signature (`SEL`), structure (`Method`), and a list of all applicable behavioural variations (`contextMethods`) of the method. Behavioural variations are created as instances of `SCContextMethod`, which are composed of their method signature, implementation, and a reference to the original method they refine. Searching for the applicable behavioural variations that can replace a method (`getApplicableMethods` in Snippet 4) requires us to traverse the list of behavioural variations (`SCContextMethods`), and select those behavioural variations for which their associated context is currently active (Lines 3-10). As many different contexts may be active, several of which may provide a different specialization for that method, an applicability order must be provided (Line 5). The exact order in which behavioural variations are applied is defined by the system's scop-

[4]A full description of method swizzling can be found in Objective-C's run-time reference: `https://developer.apple.com/library/ios/documentation/Cocoa/Reference/ObjCRuntimeRef`

ing rules (discussed later in Section 3.3.2). Once all applicable methods have been found, the second stage of the method replacement technique uses a reflective call to `method_setImplementation` (Line 20), changing the original method's implementation with that of the most appropriate behavioural variation. Upon context deactivation, a similar process is used, swizzling the original method saved in the `SCContextMethod` back to the behavioural variation currently stored in the `SCMethod`.

```
1 NS_INLINE NSMutableArray *getApplicableMethods(
      NSArray *arrayOfContextMethods) {
2   NSMutableArray *equalMethods = [NSMutableArray
        init];
3   for (SCContextMethod *contextMethod in
        arrayOfContextMethods) {
4     if ([aContextMethod isCandidate]) {
5       if ([compareMethod: contextMethod->
            specificity with: firstActiveMethod]){
6         firstActiveMethod = contextMethod->
              specificity;
7       }
8       [equalMethods addObject:contextMethod];
9     }
10  }
11  return equalMethods;
12 }

14 NS_INLINE void updateDispatch(SCMethod *meth){
15   // Search for all applicable behavioural
16   // variations for the given method meth
17  SCContextMethod *newMethodForContexts = [[
        SCMethodDispatcher sharedMethodDispatcher]
        getApplicableMethods:meth];
18  Method newImp = newMethodForContexts->method;
19   //Replace current IMP. with the appropriate
         context method IMP
20  method_setImplementation(meth->methodStructure,
        method_getImplementation(newImp));
21 }
```

Snippet 4: Objective-C implementation of method replacement for Subjective-C.

The method replacement process is shown in Snippet 4. Each new behavioural variation, `SCContextMethod` keeps a reference to the currently executing method `SCMethod` it refines, and a copy of the method implementation `Method` that it refines. Before the context is activated, an `SCContextMethod` contains its own implementation to be swizzled with the one to be refined, as depicted in Figure 3a. Whenever a context is activated, the original method implementation (`Method`) stored in the Subjective-C method (`SCMethod`) gets replaced by the new implementation of the context (`SCContextMethod`). The `SCContextMethod` itself saves a copy of the original method implementation and keeps a reference to the `SCMethod` it refines in its `refinedMethod` attribute, as shown in step ① of Figure 3b. When another context gets activated (the blue context) refining the same `SCMethod`, the method implementation stored in the `SCMethod` now becomes the `Method` implementation provided by the new context, and the `refinedMethod` of the new context is set to refer to the previous context activated, as shown in step ② in Figure 3c.

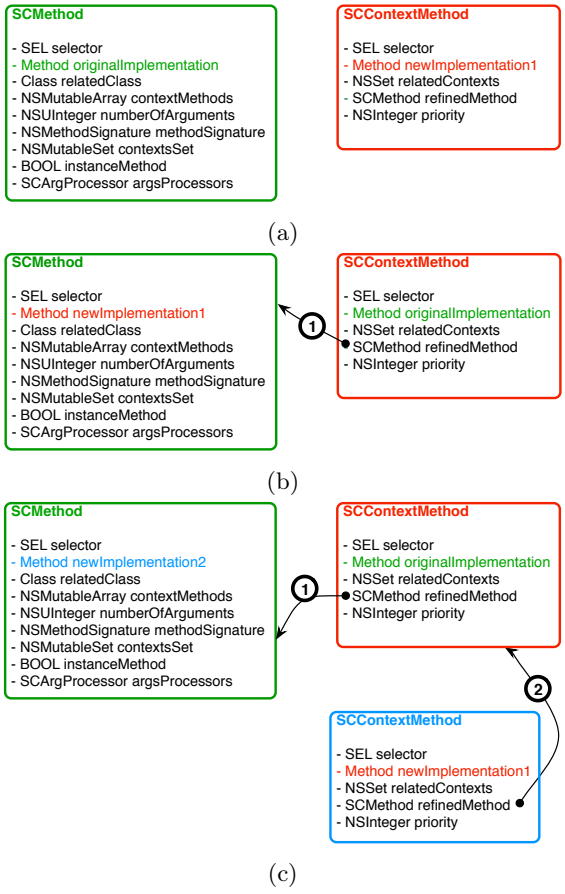


Figure 3: Method replacement and propagation

A similar method replacement technique is used in Phenomenal Gems [49] by exploiting the possibility of adapting Ruby's method tables at run time to replace a method pointer by another one. RubyCOP, a feature-based context-oriented extension to Ruby [18], implements method replacement not by changing the method table but by explicitly chang-

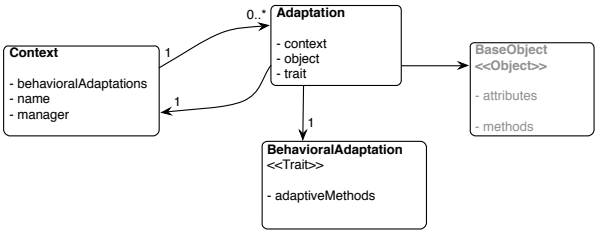
ing a method instance to a more appropriate one.

3.2.3. Fine-grained Composition

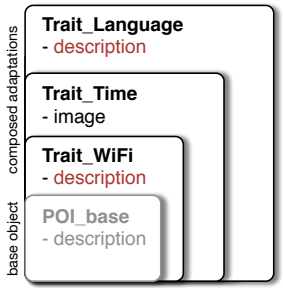
Context-oriented self-adaptive systems have two types of modules, regular modules containing the main application logic, and adaptation modules containing the behavioural variations associated to particular contexts. As context-specific situations are sensed in the environment, a new variant of the application is generated, where behavioural variations associated with the new contexts are composed with the base objects or classes they adapt. Using such an incremental composition technique requires the host language to provide a way to modify objects or classes' structure at run-time. Examples of such incremental modification mechanisms include open classes [22], mixins [59], and traits [16]. All of these mechanisms share two commonalities: (1) sparse definition of modules —that is, methods in a module can be defined at different times and locations in the system, and (2) a mechanism to resolve conflicts between matching behaviour defined multiple times. These characteristics are key to enable self-adaptivity of a software system through the run-time composition of different application modules. In this vision, program entities can be seen as aggregations of adaptations at run-time, whose behaviour can be overwritten or refined via a conflict resolution mechanism. As an example implementation of such an fine-grained composition technique, we take the trait-based approach used in Context Traits [26], a COP extension of ECMAScript, composing methods and fields to form full fletched adaptations.

Traits are the core abstraction used in Context Traits to enable run-time adaptations. Before going further, we present traits' key characteristics [16]. Trait objects can be defined sparsely, describing a set of methods they require (*i.e.,* methods without a concrete implementation) and a set of methods they provide (*i.e.,* methods with a concrete implementation). As traits are composed, required methods by one trait should be provided by another trait to complete an object. If two traits provide a same method, then a conflict resolution strategy (*e.g.,* linearization [16]) is used to specify which of the two methods will be used or applied first. Objects are said to be complete, if they have no required methods left after trait composition.

Context Traits realises behavioural variations as traits, that can be composed dynamically with ECMAScript objects (internally represented as traits) to yield adaptations. As a consequence, at any moment in time, every application object can be expressed as a composition of traits. Behavioural variations can override, extend, or reify behaviour specified by any of the application's objects, by introducing new trait method implementations. In Context Traits, behavioural variations are defined as methods in a **Trait** object,[5] while contexts are defined as standalone first-class objects. Each context object is aware of its associated behavioural variations, and each adaptation has a reference to its associated context, the object it adapts, and its behavioural variations, as shown in Figure 4a. Such associations between contexts and adaptations is achieved through the **Context.adapt(`Object`, Trait)** construct, specifying that a **Trait** object adapts `Object` whenever **Context** is sensed active.



(a) Internal entities specifying adaptations.



(b) Trait composition.

Figure 4: Dynamic adaptation through component composition with Context Traits

The dynamic composition mechanism of behavioural variations and objects consists of two elements in Context Traits: a collection of traits (*i.e.,* adaptation components), and a composition policy (*i.e.,* conflict resolution mechanism between components). Whenever a context is activated or deactivated, the behavioural variations (traits) associated with such context are composed with the base

[5] Context Traits uses traits.js, an implementation of traits for ECMAScript [63].

object they adapt. Snippet 5 shows the composition process after a context activation; the process in the case of context deactivation is similar. Context activation and management is arbitrated by a context manager (**cop.Manager**). The context manager keeps an up-to-date list of behavioural variations associated with the *currently active contexts* (Line 3). To adapt base objects, we take into account only those behavioural variations adapting the same base object the new variation adapts (Lines 4-6). Finally, these behavioural variations are ordered using the conflict resolution policy defined for the involved contexts (*e.g.,* giving priority to the adaptations activated the latest), extending the original base object with the most relevant behavioural variation (Line 11). Figure 4b shows an example of multiple traits composed with a base `POI` object in the smart city guide application. Take two active contexts `WiFi`, and `Time`, which respectively adapt the `POI` base object with the `Trait_WiFi` and `Trait_Time` behavioural variations. If we activate the `Language` context, adapting `POI` with `Trait_Language`, the resulting composition of `POI` will be `POI` = `POI_base` ▷ `Trait_WiFi` ▷ `Trait_Time` ▷ `Trait_Language`.

```
1  //Context manager: adaptation composition
2  deployAdaptation = function(adaptation) {
3     this.bAdaptations.push(adaptation);
4     relevantAdaptations = this.bAdaptations.
          filter(adap => {
5       return adap.object === adaptation.object;
6     });
7     if(relevantAdaptations.length == 0)
8       throw new Error('No adaptations found for $
            {object}');
9     this.policy.order(relevantAdaptations);
10    var adap = relevantAdaptations[0];
11    us.extend(adap.object, Object.create({},
          adapt.trait));
12 }
```

Snippet 5: Adaptation composition implementation for Context Traits.

Note that multiple behavioural variations may provide an implementation for the same behaviour. This yields a composition conflict, where it is unknown which of the implementations should be used in the final composed object. In Figure 4b, the `description` method is provided by two behavioural variations, creating a composition conflict. As with traits, a conflict resolution policy can be defined between behavioural variations to dictate which of the conflicting methods will be used at run time (Line 9 in Snippet 5). A composition policy can be seen as a function that, given a set of behavioural variations (traits), provides a linear ordering of them. In Context Traits, the default resolution strategy is to use the activation age of contexts, which gives precedence to behavioural variations associated with the contexts activated most recently. This means that in a situation where two contexts offer a behavioural variation for a same method, the variation used will be that defined by the context activated last. Other composition policies can be defined as required using the **cop.Manager.addObjectPolicy**`(Obj, Ctx, ordering)` abstraction.

ContextL is strongly based on the multiple dispatch mechanism of CommonLisp, as discussed before. However, this approach also introduces concepts of mixins to compose adaptations at run time [14]. SCROLL [42] uses a similar composition technique to the one presented here, as a means to enable dynamic adaptation of application behaviour. The implementation of SCROLL relies on traits (as a dynamic composition mechanism), and their own dispatching mechanism to direct method calls according to the active roles.

3.3. Adaptation Selection and Scoping

Up to this point we discussed *how* to dynamically adapt a software system's behaviour upon changes to its execution context. Two additional aspects important to achieve self-adaptation are: *when* to *select* the adaptations to be applied and *what* is the *scope* of applicability of these adaptations.

3.3.1. Selection

Adaptation selection refers to *when* adaptations are available to the system. There are two key moments during self-adaptive systems' execution to select adaptations: (1) whenever a call to the adapted behaviour is made, or (2) whenever its context gets activated.

Selection of adaptations during *method dispatch* (1), requires the system to select the most appropriate behaviour from all available behavioural variations in the system, upon every method call. Given that this selection mechanism relies on method dispatch, it is most suitable for COP languages implementing a *context-dependent method dispatch* mechanism. Whenever a message is sent, the concrete method implementation to call is selected

among the available behavioural variations (*i.e.,* associated to active contexts). This resembles the multiple dispatch mechanism (Section 3.2.1).

Figure 5a shows the selection of the behavioural variation available when the contexts `@user`, `@wifi`, and `@night` are active. The other variations are not chosen, as not all of their contexts are active. If there are multiple behavioural variations to choose from, as shown in Figure 5b, the behavioural variation selected corresponds to the one deemed most appropriate by the method lookup mechanism. Figure 5b, represents a situation where the `@child` context is active, the selected adaptation is the one with the `@child`, `@wifi` and `@night` contexts as the `@child` context is more specific than `@user`, following the solid line arrow.

Different policies can be defined to implement the lookup mechanism, ordering behavioural variations according to their appropriateness. A common method used for lookup is the specificity of methods' arguments, as shown in our example. The selected method is the one whose arguments are most specific with respect to the parameters in the message call. Other methods to select behavioural variations follow the delegation links for each of the method's arguments. Other policies used in the implementation of COP languages to select behavioural variations include priorities defined in context objects (*e.g.,* ContextL, Subjective-C), and context activation timestamps (*e.g.,* Ambience, Context Traits).

Selection of adaptations at *context activation time* (2), implies that all behavioural variations associated to a context are made available to the system as soon as there is a change in a context's state (*i.e.,* active ↔ inactive). COP languages implemented using a *method replacement* or *fine-grained composition* mechanisms are more suited for selection at context activation time. This is because, to implement this selection mechanism, there should be an independent entity (a *context manager* [25, 26, 46]) that orchestrates context activation and deactivation requests, that responds to such requests, assuring the system behaviour gets adapted appropriately.

The context activation time selection mechanism is implemented by connecting the context sensing component (which obtains sensor information signaling context activation or deactivation), and the adaptations in the system, through the context manager. The context manager contains a reference to all adaptations defined in the system (*i.e.,* active and inactive contexts with their associated behavioural variations). Context activation requests are processed by making all behavioural variations associated to that context available in the system. Context deactivations are treated similarly by making unavailable all behavioural variations associated with that context. For example, using a method replacement mechanism, like in Subjective-C, behavioural variations are made available by replacing method implementations for the behavioural variations associated to the context as shown in Snippet 6. Methods are selected using the function `updateDispatchedMethod`, which implements method swizzling as shown on Lines 14-21 in Snippet 4.

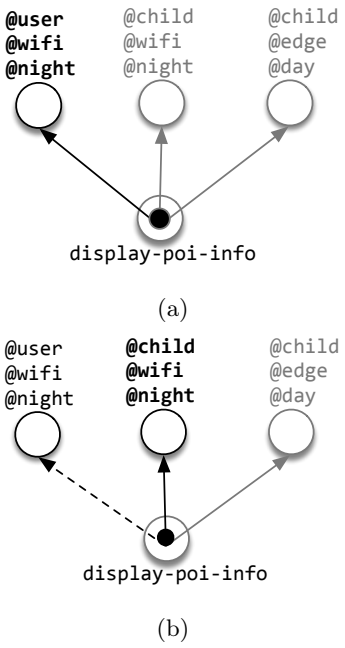


Figure 5: Adaptation selection during method dispatch

```
1 -(void) processActivationOfContext: (SCContext *)
      context {
2   [contextMethodFinder
        updateWithContextActivation: context];
3   NSSet *originalMethods = [
        originalMethodsByContext objectForKey:
        context->name];
4   for (SCMethod *orgMethod in originalMethods) {
5     updateDispatch(orgMethod);
6   }
7   [context activate];
8 }
```

Snippet 6: Selection of adaptations at context activation time in Subjective-C.

Using a context activation time selection technique, all behavioural variations associated to a context are installed as soon as the context gets activated, therefore, these will be the behaviour of the system selected on subsequent method calls.

3.3.2. Scoping

Scoping of adaptations is closely related to context activation, managing the parts of the system that are affected by a context activation, and the scope of applicability of their behavioural variations in the system. Adaptations can be scoped (1) globally to all entities of the system, or (2) locally, to a specific system entity (or process).

So far, we have considered adaptations as taking place globally. Whenever contexts are activated, their associated behavioural variations become available to all system entities. Whenever contexts are deactivated their associated behavioural variations are no longer available to any of the system entities.

Nonetheless, adaptations can be scoped locally to specific system entities such as: execution threads, object instances, or the dynamic scope of execution blocks. Locally scoped techniques (regardless of the specific entity affected by adaptations) are implemented using the dynamic characteristic of the language for dispatching the system's behaviour. In this case, the system must maintain an oracle (*e.g.,* context manager) managing the references to the local entities to be adapted.

As an example implementation of locally scoped contexts we use Subjective-C, which allows adaptations to take effect on specific execution threads. To manage local activations to a thread, the context manager maintains a dictionary for each thread, containing all context activations and deactivations for each specific thread. Local adaptations are triggered by a special construct specifying a context to activate/deactivate, and an execution thread to adapt, `@activate(ctx in thread)` and `@deactivate(ctx in thread)`, respectively. This is processed by the context manager as shown in Snippet 7. Since different threads may request to modify a context's state (*i.e.,* active or inactive) concurrently, activations and deactivations are organized using a synchronous queue system for each thread, processing the requests one by one. The first step of the activation process is to generate the activation action for the context (Lines 3-5). The second step is to retrieve the activation queue in order to add the new activation to it (Lines 6-12). Finally, the activation action created in the first step is added to the queue (Line 15), and the behaviour is updated with the new behavioural variations, using the context activation time selection technique, and the method swizzling replacement mechanism.

```
1 -(void) activateContext:(SCContext *) context
      forThread:(NSThread *) thread {
2  SCMutableQueue *activationQueue;
3  static const ActionBlock SCActivationAction = ^(
       SCContext *context) {
4    [context activate];
5  };
6  @synchronized (thread) {
7    activationQueue = [[thread threadDictionary]
         objectForKey:SCThreadActivationQueue];
8    if (!activationQueue) {
9      activationQueue = [[SCMutableQueue alloc]
           init];
10     [[thread threadDictionary] setObject:
           activationQueue forKey:
           SCThreadActivationQueue];
11   }
12 }
13 @synchronized (activationQueue) {
14   SCAction *activation = [[SCAction alloc]
         initWithObject:aContext actionBlock:
         actionBlock];
15   [activationQueue addObject:activation];
16   prepareOriginalMethodsForContext(context);
17 }
18 }
```

Snippet 7: Thread specific adaptations in Objective-C.

Similar to the local scoping of adaptations to execution threads, other COP languages scope adaptations to the lexical scope of the execution blocks in which contexts are activated (*e.g.,* ContextL), or to particular object instances (*e.g.,* EventCJ [33], ContextJS [43], and Context Traits).

4. Evaluation

Section 3.2 presents three different dynamic behaviour composition mechanisms for the implementation of internal adaptation engines from a COP language perspective. Additionally, in Section 3.3 we discussed some trade-offs regarding the adaptation selection mechanism depending on the chosen composition mechanism.

This section presents a thorough evaluation of the different implementation mechanisms to assess their applicability as a viable alternative to foster self-adaptive systems. The proposed evaluation

analyses the expressiveness, modularity, and performance of COP approaches versus the use of more direct implementation techniques to realise adaptations using internal adaptation engines in self-adaptive systems. Expressiveness is evaluated by comparing the implementations, provided by the authors, of the smart city guide case study using the three COP languages representatives of the techniques discussed in Section 3, and corresponding implementations using the strategy design pattern in each base language, as exemplars of internal adaptation engines. For each of these implementations we compare the overall system size measuring the Lines of Code (LOC), as well as the percentage of code corresponding to the application logic, adaptation logic, and adaptation management concerns as a measure of developer effort in developing each of the concerns. Additionally, we evaluate the modularity of the implementations (*i.e.,* how independent or scattered is the adaptation logic from the application logic) to observe if COP implementations do indeed satisfy the desired characteristics of self-adaptive systems. Performance is evaluated using a mini-benchmark to compare each of our COP and native implementations. First we evaluate the performance of each COP implementation against a more direct implementation of a self-adaptive system (using if-statements or the strategy design pattern). Second, we compare the performance of the different adaptation selection mechanisms offered by the COP language implementations. The purpose of this evaluation is to observe if COP approaches can scale to the size of real-world self-adaptive applications (*e.g.,* smart city applications), and to evaluate the trade-offs for the applicability of the different techniques.

All experiments were performed on a machine with a 2.8 GHz Intel Core i7 quad processor, 16GB RAM, running on OSX10. The COP languages used for the evaluation were taken from our prior experience in developing self-adaptive systems, and are representative of existing COP languages and techniques (*cf.,* Section 5). Ambience [23] is used as a language featuring context-dependent method dispatch, Subjective-C [25] is representative for the method replacement mechanism, and Context Traits [26] exemplifies fine-grained composition mechanisms. The host languages for each of these representative languages are, respectively: Common Lisp, Objective-C and ECMAScript.

4.1. Expressiveness and Modularity

The purpose of evaluating the expressiveness of internal adaptation engines is to support our claim that, when appropriately designed and implemented, these approaches are **modular**, **concise**, **fine-grained**, and **understandable**. We can distinguish three main concerns in the development of self-adaptive systems: application logic (■), adaptation logic (■), and adaptation management (■). *Application logic* refers to all code required to implement the system's main functionality. *Adaptation logic* refers to the functionality to define adaptations in the system –that is, behavioural variations and their associated contexts. Finally, the *adaptation management* concern refers to the functionality needed to orchestrate adaptations as they are introduced to, or withdrawn from the system. Our hypothesis is that COP languages limit the amount of *adaptation management* needed, and that they better separate the *application logic* from the *adaptation logic*, reducing the effort required to introduce adaptations in a software system.

In this section we compare the expressiveness of self-adaptive systems with internal adaptation engines implemented using the chosen COP languages, and those implemented natively in the host languages using a common OOP pattern like the strategy design pattern that enables the modification of behaviour at run time. In the evaluation we focus on the conciseness and granularity of adaptations measuring the effort required to implement the smart city guide case study as the LOC used for each concern. We compare the two versions of the case study, for each of the languages and their underlying implementation techniques.[6] Additionally, we measure the modularity and understandability of the case study taking into account how cleanly separated are the application, adaptation, and management concerns.

For each of the chosen languages, code snippets 8-13 show each of the three concerns in the implementation of the `English` context for the smart city guide, and the associated behavioural variation of the `name` functionality for `POI` objects. The dotted lines in each snippet correspond to different modules in which the desired functionality is implemented.

From these snippets we observe how the three concerns in the implementations based on the strat-

[6] All implemented versions of this case study are available at: `https://github.com/ncardozo/SmartCityGuide`

```
(defclass @poi ()
  (name))
(defun base-name-strategy ((poi @poi))
  (error 'language-error :text "A language needs to be defined to
        get the name"))
-----------------------------------------------------------------
(defun english-name-strategy ((poi @poi))
  (sql-select :columns (list "name")
       :tables (list "Poi")
      :where (sql-binary-operator :name '= :left (sql-identifier
      :name "langauge")
                :right (sql-identifier :name "EN"))))
-----------------------------------------------------------------
(defun base-set-language-strategy ((cm @contextManager) (lang))
  (setf currentLanguage lang)
  (update-language cm))
(defun base-update-language-strategy ((cm @contextManager))
  (cond ((eq 'currentLanguage "EN") (setf *lang-context* "english
      -"))))
```

Snippet 8: Strategy-pattern adaptation in Common Lisp

```
(defproto @poi (clone @object))
(add-slot @poi 'name "")
(defmethod name ((poi @poi))
        (error 'language-error :text "Language definition needed
              to get the name"))
-----------------------------------------------------------------
(with-context (@english)
  (defmethod name ((poi @poi))
   (sql-select :columns (list "name")
      :tables (list "Poi")
      :where (sql-binary-operator :name '= :left (sql-identifier
      :name "language")
                  :right (sql-identifier :name "EN"))))
-----------------------------------------------------------------
(defcontext @english)
-----------------------------------------------------------------
(defmethod update-language ((cm @contextManager))
 (cond ((eq 'currentLanguage "EN") (activate @english)))
```

Snippet 9: Language adaptation in Ambience

```
var PoiStrategy = function(nid) {
  this.id = nid;
};
PoiStrategy.prototype.getName = function() {
  throw new Error("Language strategy needed to get the language
        name");
};
-----------------------------------------------------------------
var Poi = function(name, strategy) {
        this.strategy = strategy;
        this.strategy
            .setName(name);
};
Poi.prototype.setStrategy = function(strategy) {
        this.strategy = strategy;
};
Poi.prototype.name = function() {
  return this.strategy
             .getName();
};
-----------------------------------------------------------------
var EnglishPoiStrategy = function(){};
EnglishPoiStrategy.prototype = Object.create(PoiStrategy.
      prototype);
EnglishPoiStrategy.prototype.getName = function() {
  var res = db.runQuery('SELECT name FROM Poi WITH language = %@
      AND id_poi = %d', 'EN', this.id);
  return res[0];
};
-----------------------------------------------------------------
ContextManager.prototype.updateLanguage = function() {
  if(this.currentLanguage == "EN")
    POI.Poi.setStrategy(new POI.EnglishPoiStrategy());
};
```

Snippet 10: Strategy-pattern adaptation in ECMAScript

```
TPoiOperations = Trait({
  getName: function() {
   throw new Error("Language definition needed to get the name");
  },
});
function TPoi(name) {
  return Trait.compose(TPoiOperations,Trait({ name: name }));
}
function Poi(name) {
  return Object.create(Object.prototype, TPoi(name));
}
-----------------------------------------------------------------
TEnglishPoiOperations = Trait({
  englishName: "",
  getName: function() {
    var res = db.runQuery('SELECT name FROM Poi WITH language = %
      @ AND id_poi = %d', 'EN', this.id);
   return res[0];
  },
});
-----------------------------------------------------------------
English = new cop.Context({ name: "English"});
English.adapt(POI.TPoiOperations, POI.TPoiEnglishOperations)
-----------------------------------------------------------------
ContextManager.prototype.updateLanguage = function() {
  if(this.currentLanguage == "EN")
    English.activate();
};
```

Snippet 11: Language adaptation in Context Traits

```
@interface BasePoi : NSObject {}
@interface PoiEN : BasePoi
 -(NSString *) name;
@end
@interface Poi : BasePoi { ... }
@implementation Poi
  NSString *name;
  id strategy;
-(NSString*) name {
  return [NSString stringWithFormat:@"INIT_ERROR-%@", @"name"];
}
+(void) setStrategy:(id)_strategy {
  self.strategy = _strategy;
}
+(id) getStrategy {
  return self.strategy;
}
@end
-----------------------------------------------------------------
@implementation PoiEN
-(NSString *) name {
  NSString *query = [NSString stringWithFormat:@"SELECT name
      FROM Poi WITH language = %@ AND id_poi = %d", @"en", [self
      poiId]];
  NSArray *res = [[NSArray alloc] initWithArray:[[self dbManager]
       loadDataFromDB:query]];
  return [res objectAtIndex:0];
}
@end
-----------------------------------------------------------------
@LanguageViewController
-(void) updateLanguage {
  [POI setStrategy:[[NSClassFromString([NSString stringWithFormat
      :@"Poi%@", newLanguage]) alloc] init]];
}
@end
```

Snippet 12: Strategy-pattern adaptation in Objective-C

```
@implementation Poi
NSString *name;
-(NSString*) name {
  return [NSString stringWithFormat:@"INIT_ERROR-%@", @"name"];
}
@end
-----------------------------------------------------------------
@implementation Poi (Poi_EN)
@contexts English
-(NSString *) name {
  NSString *query = [NSString stringWithFormat:@"SELECT name FROM
       Poi WITH language = %@ AND id_poi = %d", @"en", [self poiId
      ]];
  NSArray *res = [[NSArray alloc] initWithArray:[[self dbManager]
       |loadDataFromDB:query]];
  return [res objectAtIndex:0];
}
@end
@end
-----------------------------------------------------------------
@implementation ContextManager
 @context(English);
@end
-----------------------------------------------------------------
 self.currentLanguage = "EN";
@LanguageViewController
-(void) updateLanguage {
  @activate(currentLanguage);  //new language contains the
      current language e.g., EN
}
@end
```

Snippet 13: Language adaptation in Subjective-C

egy design pattern (Snippets 8, 10, 12) are more scattered across the different modules, than in the case for their corresponding COP counterparts (Snippets 9, 11, 13), which respect module boundaries more strictly. This gives a first indication that the COP implementations are more modular and maintainable.[7]

The COP implementations highlight three properties that evidence the expressiveness, understandability, evolvability and maintainability of COP-based adaptation engines: (1) The signatures of behavioural variations are the same as for the definitions of the base behaviour (*e.g.,* `name()`). That is, it is not required to introduce additional infrastructure for the definition of behavioural variations, which we remark as a particular advantage of expressing adaptations using COP; (2) The definition of the `English` adaptation does not involve the direct modification of any application logic. Rather, it is implemented as an independent module. This opens up the system to dynamic behaviour adaptation, as new or updated functionality can be introduced as independent context-specific adaptations/modules. This improves system maintainability, as maintenance tasks are local, not requiring modification of the adaptation or application logic across already existing modules. (3) The modular approach to define adaptations (*i.e.,* declaring them in independent implementation modules from the application logic) induces a better separation of concerns between the base and adaptation logic.

Additionally, comparing implementations we note that the COP-based implementations are often more **concise** (less LOC), more **modular** and **finer-grained** than those using pattern-based implementations. This can be explained by two phenomena. (1) First, the difference between the two implementation approaches comes from adaptations' modularity. In the strategy pattern implementations, the variation associated to the `English` context is defined by a new system module, `PoiEN`, extending the original `Poi` object through a hierarchical relation (*i.e.,* in Objective-C the inher-

[7] The term *scattering* stems from Aspect-oriented Programming [40], and refers to the implementation of concerns of which the code is spread out over multiple modules. Such concerns affect the implementation of multiple modules, which is bad for modularity.

itance relation to the `BasePoi` object). The definition of the contexts associated to behavioural variations are global and do not require the definition of any additional objects. All contexts are defined in the same module. As no additional base objects are required in the definition of adaptations using COP languages, the adaptations themselves are introduced at a finer level of granularity (*i.e.,* methods). (2) Second, using a pattern-based internal adaptation engine, the management of adaptations requires the definition of an additional module to relate all adaptations with the base object. Additionally, adapted objects' application logic is polluted, adding properties and behaviour not inherent to them. In contrast, COP languages manage adaptations internally, hiding the intrinsic details of adaptation interactions to developers. The system logic only requires to use the context activation and deactivation API offered by the language. Such characteristics make these approaches for internal adaptation more modular and understandable, as all (and only those) concerns related to an object's behaviour are encapsulated within the object itself, while all adaptation concerns are encapsulated in context objects, behavioural variations, and the adaptation manager (in different modules).

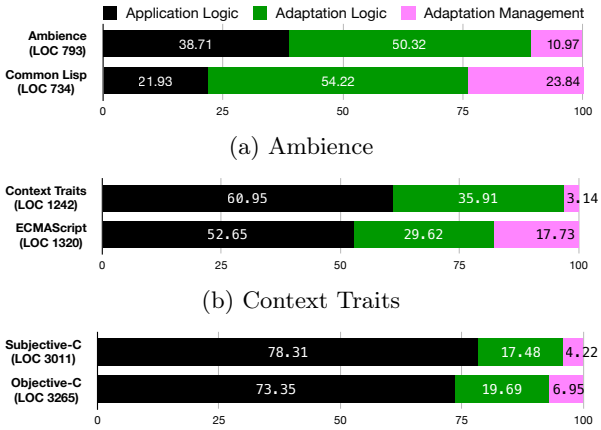


Figure 6: LOC percentage dedicated to each concern in the smart city guide application

Figure 6 shows the percentage of LOC for the application logic, adaptation logic, and adaptation management concerns for each of the six implementations of the smart city guide. The figures exhibit the development effort required to implement each of these three concerns of self-adaptive systems.[8]

Figure 6 (6a, 6b, 6c) shows a clear reduction in the percentage of code related to the adaptation management concerns across all languages. In the case of Subjective-C, in Figure 6c, the improvement of the adaptation management logic is less pronounced as this language uses additional definitions for context dependency relations, not present in the other languages. This increases the amount of adaptation management required. Nonetheless, context dependency relations are defined through a DSL [25], and therefore the effort for developers is reduced. Note, however, that the code of the DSL is not taken into account in the LOC measurement, we only take into account the final Subjective-C code. The figure also indicates an increase in the percentage of application logic and a decrease in the percentage of adaptation logic when using COP-based approaches. This is due to the increased modularity and reduced tangling of application and adaptation logic, the implementation of the adaptation logic gets much more concise, thus increasing the overall percentage of application logic correspondingly. This suggests that the use of well-designed internal adaptation engines, improves the development of self-adaptive systems by reducing the size of the adaptation logic and delegating the complexity of managing adaptations to the adaptation engine. This helps developers to focus on the main application logic. For both Ambience and Context Traits, we observe and improvement between 8% to 17% in the code base dedicated to the application logic together with its related adaptations. In the case of Context Traits, we note a large reduction in the adaptation management code, with a slight increase in the percentage of adaptation logic. Such increase is due to the underlying infrastructure (*i.e.,* traits) used to define behavioural variations.

A final observation from Figure 6 is that, apart from the case of Ambience (6% increase with respect to CLOS), the total LOC of the system decreases by more than 10% for the COP implementations, pointing to a more concise implementation that is easier to understand, maintain, and

[8]The percentage of application logic is significantly higher for the Objective-C and Subjective-C implementations, as these implementations include the full user interface, to display the application on a mobile phone. This code was simplified in the other implementations by using a text-based interface to interact with users.

evolve (especially taking into account that the implementation becomes more modular and contains less adaptation management code).

With the evaluation of the expressiveness of COP approaches, we can conclude these approaches are beneficial to realise self-adaptive systems, that are concise and maintainable, as opposed to common beliefs about these approaches.

Figure 7 shows our evaluation of the modularity between the different self-adaptive implementations of the smart city guide case-study. The figure indicates the amount of tangling in the definition of the applications between our pattern-based and COP-based implementations. Here we show the number of classes in which each adaptation concern is tangled. The more adaptations a class is intertwined with, the less modular the code is. In the case of Objective-C, 5 classes implement a single adaptation, but there are also classes tangled with 5 or 6 different adaptations. Similar things happen for the amount of tangling that can be observed for both Common Lisp and ECMAScript where only a few adaptations are not tangled over multiple classes, but several are tangled over multiple classes. In all implementations, we identify 1 class tangled with 12 different adaptations. This class corresponds to the context manager, where all contexts and the conditions for their activation are defined. In general however, the corresponding COP implementations present less tangling. In the case of Subjective-C and Context Traits in particular, there is a significant increase in the number of modules with only one adaptation, showing that in these cases we achieve near-perfect modularity as nearly all of the adaptations are contained in an independent module.

We can thus conclude that COP-based internal adaptation engines do improve the system modularity and reusability.

4.2. Internal Adaptation Engines Performance

We now turn our attention to a performance evaluation of each of the language implementations, based on two criteria. First, we compare the overall performance of each of the COP-based internal adaptation engine implementations against adaptations implemented directly within the host programming language. Our objective is to evaluate that COP-based implementations do not deteriorate performance at scale. Second, we compare the performance of the different selection mechanisms for behavioural variations. The objective of that evaluation is to present the performance overhead of the different approaches to select behavioural variations so that the most appropriate one can be chosen for a particular type of self-adaptive system.

4.2.1. COP vs Native Adaptation Implementations

To compare the performance of COP approaches and internal adaptation engines we use a mini benchmark [25]. The purpose of the benchmark is to evaluate the impact of using behaviour variations with the different adaptation strategies and selection mechanisms. The benchmark consists of calling a `test` method repeatedly within different contexts (all specializing the same `test` method), as specified in Snippet 14.

The benchmark measures the time it takes to activate one out of a set of possible contexts, call the `test` method 100 times for each context activation and then deactivate that context. The performance of these method calls is measured for context sets of ten different sizes (*i.e.,* 10, 50, 100, 500, 1000, 2500, 5000, 10000, 25000, and 50000 contexts). Each instance of the benchmark (*i.e.,* calling the test method for a number of contexts) is executed 10 times.[9]

```
-(double) methodCallBenchmark {
  NSDate *start = [NSDate date];
  for(int i = 0; i < 1000; i++) {
    int num = 1 + arc4random_uniform(
        NUM_OF_CONTEXTS);
    @activate([NSString stringWithFormat:@"
        Context%d", num]);
    for(int j=1; j<=100; j++)
       [self test];
    @deactivate([NSString stringWithFormat:@"
        Context%d", num]);
  }
  NSTimeInterval timeInterval = fabs([start
      timeIntervalSinceNow]);
  return timeInterval*1000;
}
```

Snippet 14: behavioural variations mini-benchmark in Subjective-C.

Figure 8 shows the results for our three COP language implementations (*i.e.,* Ambience, Context Traits, and Subjective-C). We compare the benchmark results for each of these COP implementations (▲) with a native implementation that specialises

[9] All benchmarks results are available at: `https://github.com/ncardozo/CopPerformanceEvaluation`

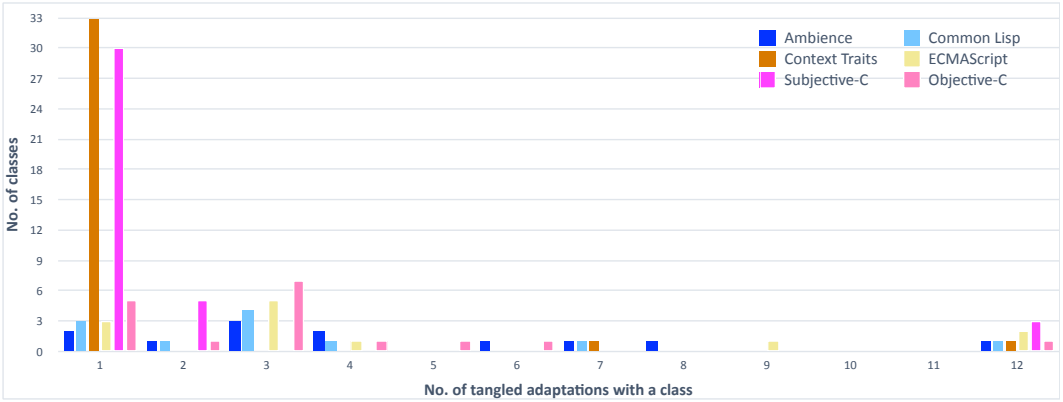


Figure 7: Density of tangled adaptation concerns per class.

behaviour directly by means of `if`-statements (■), and an implementation that makes use of the strategy design pattern (●).

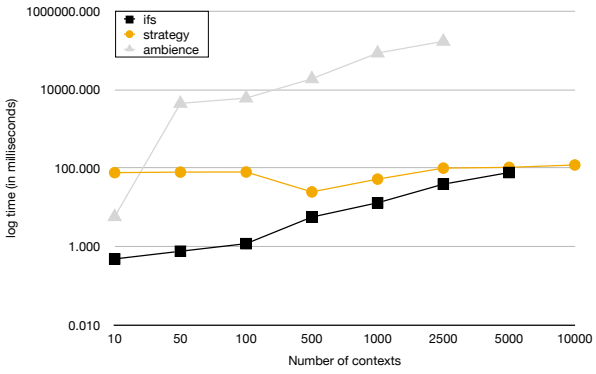
The results in Figure 8 show the average time (in logarithmic scale) over 10 benchmark runs. Note that each graph shows the largest context set size for which at least one implementation terminates successfully. The if-case implementation yields an error in Objective-C and Common Lisp when running 10000 `if`-cases, while it took ECMAScript 5000 `if`-cases to run into an exception. The strategy implementation throws an error in CLOS with 25000 strategies. The COP implementation throws an error in Ambience with 5000 contexts. The Subjective-C implementation is the only implementation to finish all cases.

Figure 8 illustrates the performance cost of using (many) adaptations. This cost is due to the additional step for the introduction and withdrawal of adaptations required in such approaches. In all cases, for a small number of adaptations, the if-case implementation is the most performant. However, as the number of adaptations increases if-cases get outperformed by the strategy and COP implementations (except for the case of Ambience), as the number of required conditional checks increases. We also observe that the strategy-based implementation eventually performs best for all three implementations. This occurs because of the unique combination between fixed dependencies and dynamic method forwarding used in this pattern. At the beginning of a run using the strategy design pattern all dependencies between objects are known and available to the system (similar to the dependencies generated with if-cases). At run time, only one of these dependencies is used, changing the strategy globally in the application, which gives direct access to the behavioural variations implemented in such a strategy (similar to the global selection mechanism in COP). In contrast, if-cases need to be re-checked for all possibilities for every method call which explains why these implementations have a higher run-time performance overhead.

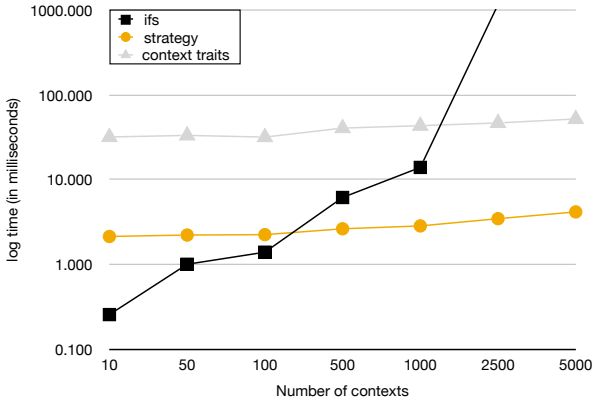
From all COP implementations, Context Traits offers the best performance, albeit a factor 10 slower than its corresponding strategy pattern-based implementation, while still providing the added value of being more flexible and modular regarding the definition and introduction of adaptations. Moreover, we note from our implementations that both the Context Traits and Subjective-C implementations are able to finish the benchmark, while the other internal adaptation engine implementations are not always able to complete the benchmark. There is a significant performance impact of COP approaches. While this approaches are still able to work properly in scenarios that excede the number of adaptations in real-world self-adaptive, more research in improving their performance is required.

4.2.2. Selection Mechanisms' Performance

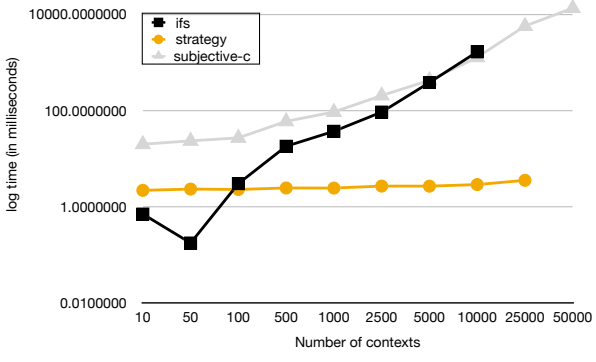
From our simple benchmark we can observe that the type and domain of self-adaptive systems may influence the decision of what implementation technique or language best to use. In particular, this benchmark evaluates a scenario where method calls occur more frequently than context changes. As such, it favours COP approaches implementing a *context activation time* selection mechanism, like the one in Context Traits or Subjective-C. With such a mechanism, all behavioural variations are enacted as soon as there is a context change that



(a) Ambience



(b) Context Traits



(c) Subjective-C

Figure 8: Performance comparison between all COP-based implementations

triggers them, making these behavioural variations readily available as soon as they are called, with no additional dispatching. Domains in which the context changes are more frequent than the operations within, benefit of a *method call* selection mechanism, like the one in Ambience, in which the behavioural variation is chosen only when they are needed.

4.3. Discussion of the results

We now discuss the results of the different COP approaches and their underlying techniques, as explored in this paper. This section puts into perspective the analysis of the appropriateness of these approaches as a means to implement effective internal adaptation engines for self-adaptive systems discussed in Section 4.4.

With respect to the modularity evaluation, we observe that the approaches with a better separation of concerns are those of Ambience (context-dependent method dispatch) and Context Traits (fine-grained composition), while Subjective-C (method replacement) still presents some cross-cutting concerns. When evaluating the expressiveness of these two approaches, we observe, that there is no conclusive winner, and both approaches present a similar improvement with respect to the non-COP-based implementations. The context-dependent method dispatch mechanism requires more attention to context activation and deactivation, as well as the focus on the adaptation logic, while the impact of these aspects for the fine-grained composition method is a bit lower.

Adaptations can be realised by coupling the adaptation logic with the main application logic, for example, through if-cases or the use of software design patterns. Nonetheless, the performance evaluation in Section 4.2.1, shows how the design pattern-based implementations outperformed the other two implementations. The if-case implementations were the most problematic in most cases, as their performance decreases rapidly across all languages, not being able to complete the benchmark execution due to memory overflows. The performance of COP languages also decreases rapidly in the case of Ambience (context-dependent method dispatch) and Subjective-C (method replacement). Such poor performance may be due to the underlying structures and algorithms used for each language. Ambience relies in the C3 algorithm to resolve the method called taking into account the contexts. As the amount of contexts increases, the time to execute the context change does as well. As similar phenomenon occurs in Subjective-C, as the number contexts increases, the time to activate and deactivate the context does so as well, as it is necessary to go through all contexts to detect the method to execute. The case of Context Traits (fine-grained

composition), however, does not exhibit a significant performance decline as the composition of the adapted method is executed directly on the activated traits, therefore the number of contexts in the system is not relevant. When managing a small set of contexts, is better to opt for a method replacement or context-dependent method dispatch approach. If the number of contexts in the system increases, a fine-grained composition method should be preferred.

In addition to the aforementioned characteristics, we can also consider the type of system to implement in order to choose the most appropriate approach. As mentioned in Section 4.2.1, if adaptations are expected to take place more frequently than method calls, then a method dispatch selection mechanism may be preferable. If the opposite is the case, an adaptation time selection mechanism should be preferred. The advantage of internal adaptation engines, as conceived by COP languages, is that both methods could be combined into a single adaptation engine [43, 10, 34].

4.4. COP as an Implementation of Self-adaptive Systems

We now analyse the self-adaptive properties that COP-based internal adaptation engines exhibit, and discuss how these properties are supported by the results from our evaluation.

Adaptation engines render software systems **smart**, **open**, and **customizable** by enabling dynamic introduction and withdrawal of behavioural variations in response to changes in a system's internal state or surrounding environment. This paper advocates internal adaptation engines by presenting COP approaches implemented using different techniques that offer better software engineering characteristics (*e.g.,* modularity, granularity, conciseness, and understandability) in the design and implementation of adaptive systems, over other traditional internal adaptation engine implementations. The appropriateness COP-powered internal adaptation engines is illustrated through the multiple implementations of our smart city guide case study.

From the expressiveness evaluation in Section 4.1, we observed that COP-based language implementations provide better **modularity** than the implementation based on design patterns, as they reduce the scattering and tangling of adaptations across modules, without the additional architectural footprint.

COP also renders self-adaptive systems more **concise** and **understandable** by hiding the complexity of defining and managing adaptions inside the internals of the adaptation engine (*i.e.,* the programming language). The decoupled definition of adaptations (in contexts and behavioural variations) also enables the definition of very **fine-grained** adaptations (up to the level of individual methods), as opposed to the component-level adaptation normally used in self-adaptive systems.

Three properties of self-adaptive systems are not explicit in our evaluation as they do not pertain the implementation of COP languages, but are intrinsic properties COP languages provide. The **extensibility** provided by internal adaptation engines can be considered an extension of the basic adaptation process. COP-based adaptation engines have been used to extend systems' functionality through adaptations, which render them (1) **highly-available**, as the system does not need to be stoped to incorporate new features or bug-fixes, and (2) **resilient**, as COP languages, independent from their implementation, can manage the interaction between adaptations to avoid undesired behavior [8].

As a consequence, COP-based language implementation techniques for internal adaptation engines are proven **flexible**, **modular**, and **reusable** for developing a wide range of self-adaptive systems.

4.5. Threats to Validity

In this section we discuss the threats to the validity of our evaluation.

Threats to external validity refer to the generalization of the results from our evaluation. We explore the expressiveness, modularity, and performance aspects for the studied implementation techniques to realize COP languages. The evaluation of these aspects offers a clear view about the differences and similarities of the different implementation techniques. Such evaluation can be used to assess the appropriateness of the implementation techniques for a given base language or application domain. Note, however, that the evaluated aspects focus on the implementation issues of representative COP languages. Other characteristics of the languages could be used to assess the appropriateness of a COP language for a given domain [19]. Such characteristics could be used to complement our evaluation and offer more informed decisions on the implementation of COP languages.

Threats to construct validity refer to the cases in which the evaluation is not exhaustive or selective

enough. In our case, we evaluate the expressiveness and modularity of the system using the implementation of a large case study. The system implementations were performed by the paper authors, who have experience in building context-aware systems with the languages presented in the paper. A broader empirical study with different types of developers implementing case studies from different application domains, could confirm our results or make the difference between the language implementations more clear.

Additionally, our benchmark evaluation for both the scale of COP performance with its internal adaptation engine counterparts, and the COP selection mechanisms lets us observe the behavior of COP as an internal adaptation engine at a large scale. Even though the presented benchmarks are simple, they scale the amount of available adaptations and behavioral variations well over known self-adaptive systems exemplars.[10] While more complex benchmarks could be executed, our initial evaluation, does support our hypothesis about the usefulness of internal adaptation engines. Moreover, it posits COP approaches as a viable candidate to realise self-adaptive systems.

5. Related Work

Aside from the techniques to implement internal adaptation engines presented in this paper, there are other implementation mechanisms using dedicated languages or compiler extensions, to enable dynamic adaptation. Nonetheless, the implementation of these internal engines follow similar patterns as the ones described in Section 3. Below, we provide a comprehensive list of different techniques used to implement COP languages, giving pointers to their specific implementations for interested readers. This section follows the structure of Table 1, where we discuss first techniques most related to COP languages that fit completely (marked as ● in the table) within the categories given in Section 3, or use a combination of the techniques described in Section 3 with other techniques outside the ones explained in this paper (marked as ◐ in the table), and other possible approaches (named in the last column of the table) The second section of Table 1, and of this discussion, presents other approaches that can be used to realise internal adaptation engines using programming language approaches other than COP.

Aspect-oriented Programming (AOP) introduces a dynamic code augmentation technique that can be used to achieve run-time adaptations to the surrounding environment [50]. AOP enables the augmentation of an application's base code by associating new behaviour (*i.e.,* advices) with specific program points (*i.e.,* join points). AOP can be used to dynamically introduce new behaviour into a system, similar to the way the method replacement mechanism works. That is, at run time, the behaviour to execute depends on which advices can be applied to join points in the base code. Advices typically get woven into the system at aspect-weaving time (before execution), but some AOP compilers can enable the introduction of advices at run time, hence affecting the system's behaviour dynamically [56, 64].

To implement a COP-based internal adaptation engine with AOP two things are required: a translation of COP concepts to an aspect language (to augment the code), and a manager to dispatch the behaviour to the correct implementation. In this process, specialized adaptation abstractions, context declarations, and transition rules are translated to advices, pointcuts, classes, and methods in the aspect language. Behaviour variations can be made available to the system by introducing advices dynamically, using dynamic aspect languages such as CaesarJ [5]; or by implementing alternative method dispatch mechanisms as a technique to drive adaptations [12] (similar to the one presented in Section 3.2.1). The most prominent implementations of COP languages using AOP as an underlying thechnique are: JCop (an extension of ContextJ) [3, 4], and ServalCJ (formerly known as EventCJ) [35, 33, 36]. Both of these implementations extend the abc compiler [1] with their own dispatch mechanism. Other implementations, like the one in ContextS [29], extends Squeak/Smalltalk based on the use of metaclasses in AspectS. In particular this last approach offers a mixture of the context-dependent method dispatching and the method replacement techniques.

Predicate dispatch is an alternative to the context-dependent dispatch mechanism presented in Section 3.2.1. Predicate dispatch relies on the idea that behavioural variations are associated with

[10] https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/

Table 1: Summary of COP approaches and how they fit into the described techniques

Approach	Context-dependent method dispatch	Method replacement	Fine-grained composition	Other
ContextL [13]	◐	○	◐	○
Lambic [62]	●	○	○	○
Flute [6]	●	○	○	○
Congolo [45]	●	○	○	○
Phenomenal Gems [49]	○	●	○	○
RubyCOP [18]	○	●	○	FOP
ContextS [29]	○	◐	○	AOP
JCop [3]	○	○	◐	AOP
ServalCJ [35]	○	○	◐	AOP
ContextErlang [21]	○	○	○	Structure modification
rbFeatures [27]	●	○	○	FOP
SCROLL [42]	◐	○	◐	ROP
Object Teams [28]	○	○	◐	ROP
LyRT [61]	◐	○	○	ROP
Java Adaptor [51]	○	○	◐	Software updates
SmallTalk [66]	○	○	◐	Software updates
DuST'M [48]	○	○	◐	Software updates

a predicate expressing conditions about the internals of the system or its surrounding environment (similar to their association with contexts, as explained before). Whenever a method is called, all predicates of all behavioural variations matching the method signature are evaluated, and the behaviour selected is that corresponding to valid predicates. If multiple predicates are valid at the same time, they are executed in some order, for example by specificity of the method arguments. Predicate dispatch mechanisms have been explored in COP languages for the adaptation of group behaviour in distributed systems in Lambic [62], and the adaptation of reactive systems in Flute [6].

Structure modification Adaptation of a system's behaviour can also be achieved by modifying the internal structure of the system itself. In programming languages like Erlang, an application's behaviour is structured as a supervision tree consisting of workers (the tree leaves) monitored by a set of supervisors (internal nodes). Adaptations are realized by dynamically binding new behaviour to the supervision tree (as new tree leaves; the structure of supervisors does not change at run time). This dynamic binding process is supported by an additional *context ADT* structure managing interactions between adaptations. Specifically, when two adaptations for a same behaviour are bound to the system, these are ordered hierarchically in the context ADT. The behaviour observed in the application is given by the selection of the adaptation most specific in the context ADT. This approach has been used for concurrent systems, supported by the ContextErlang language [21, 53].

Other approaches have been suggested for the implementation of dynamic adaptations from a programming language perspective. Similar to Flute, other approaches to enable *reactive* adaptation to the context have been proposed [37, 32, 41]. Behaviour modifications in these models follow those described in this paper (the current implementations are based on AOP solutions), with the additional feature of reacting promptly to context activation, through the use of signals.

The *role-object model* [7] introduces a new modularization strategy for developing software sys-

tems. In this model, objects are defined and augmented according to the different roles they play during the system's lifespan. At run time, objects change roles dynamically, adapting their state and behaviour with respect to their new role. Adaptation of objects via roles can be achieved through internal adaptation engines integrated within a programming language. Similarly to the approaches described in Section 3, the specific mechanisms to realise Role-oriented Programming (ROP) depend on the characteristics provided by the underlying language, ranging from dynamic binding of new behaviour as in LyRT [61], run-time composition of objects as in Object Teams [28], or method dispatch techniques as in SCROLL [42].

Advances in Feature-oriented Programming (FOP) also enable the implementation of internal adaptation engines to dynamically introduce features. In this paradigm, adaptations are defined as features that can be loaded and unloaded at run time. The inclusion/exclusion of a feature in the system is achieved through method replacement mechanisms as described in Section 3.2.2. Such mechanisms are implemented as part of rbFeatures in Ruby [27], Phenomenal Gems [49], and RubyCOP, feature-based context-oriented extensions to Ruby [18].

Congolo [45] presents an approach very similar to method swizzling, exploiting the `invokedynamic` feature of Java. In Congolo, adaptive behaviour is incorporated into the running system by dynamic loading modules into the JVM.

Finally, the domain of hot software upgrades also enables the adaptation of system behaviour at run time by changing objects' or components' definition in the virtual machine. Usually, these approaches imply the complete replacement of an object or component by an equivalent one with an adapted behaviour. Many different mechanisms exist for dynamically upgrading components according to the implementation language or technique used. Examples include compiler extensions of the JVM as in Java Adaptor [51], using contexts in SmallTalk [66], or using Software Transactional Memory (STM) in Java [48].

The first COP approaches discussed in this section are directly related to those presented in Section 3, as AOP can be considered as a (reflective) method replacement mechanism, and predicate dispatch is a context-dependent method dispatch mechanism. Other approaches use different mechanisms, but can also be regarded as a kind of method replacement mechanism. Several of the properties highlighted in our approaches may also be applicable to these alternative mechanisms. A more detailed comparative evaluation (*e.g.,* Cardozo et al. [9] or Elyasaf and Sturm [19]) of these approaches would be required to fully assess the subset of self-adaptive systems properties they satisfy.

6. Conclusion

This paper advocates the use of *internal adaptation engines* as a viable option for the development of context-oriented self-adaptive systems. More specifically, we analyse the use of COP languages as a means of implementing software systems that can adapt their behaviour at run time; yielding adaptive software systems that are: smart, open, resilient, extensible, highly available, customizable, modular, fine-grained, concise, and understandable. COP languages abstract away the adaptation logic and management from developers by keeping it internal to the programming language, with the advantage, contrary to common belief, that adaptation and application logic remain cleanly separated. To achieve this, COP languages provide language-level abstractions for the definition of adaptations, in the form of contexts and their associated behavioural variations, that can be dynamically composed with, or withdrawn from, the running system in response to input from the surrounding environment. We demonstrate how to attain the aforementioned properties in internal adaptation engines by showcasing their implementation in COP languages, through different programming language-level mechanisms. The presented implementations are representative of a wide range of available implementation possibilities to enable run-time adaptation according to the facilities offered by the host language: context-dependent method dispatch, method replacement, or fine-grained composition. We evaluate the properties of these COP-based internal adaptation engines in two ways. **(1)** First, we conducted an empirical evaluation of the feasibility and applicability of internal adaptation engines at the programming language level by showcasing their use in the development of a real-world self-adaptive system, a smart city guide. We posit the expressiveness and modularity of COP-based adaptation engines, in contrast to a more straightforward implementation

in terms of the strategy design pattern. This evaluation highlights the benefits of COP with respect to increasing the expressiveness and readability of adaptations and their management, and reducing the dependencies between adaptations, as well as their scattering across and tangling with the application logic. **(2)** Second, we present a quantitative evaluation using a mini-benchmark to show the effectiveness of various COP-based adaptation engine implementations by comparing their run-time execution performance with that of more straightforward implementations in the host language. Our evaluation shows that, while managing to keep the adaptation and application logic cleanly separated, COP-based internal adaptation engines still perform quite well in highly dynamic environments, putting them in a favorable position to realise self-adaptive systems.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: The aspectbench compiler for aspectj. In Robert Glück and Michael Lowry, editors, *Generative Programming and Component Engineering*, pages 10–16, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31977-1. doi: 10.1007/11561347_2.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP'09, pages 1–6, New York, NY, USA, 2009. ACM Press. ISBN 978-1-60558-538-3. doi: 10.1145/1562112.1562118.

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. Contextj: Context-oriented programming with java. *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software*, 28(1):272 – 292, 2011.

[4] Malte Appeltauer, Robert Hirschfeld, and Jens Lincke. Deaclarative layer composition with the jcop programming language. *Journal of Object Technology*, 12:4:1–37, 2013.

[5] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Transactions on aspect-oriented software development i. chapter An overview of caesarj, pages 135–173. Springer-Verlag, Berlin, Heidelberg, 2006.

[6] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang 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*, OnWard'12, Tucson, Arizona - USA, October 2012. ACM.

[7] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. *Role Object*, chapter 2, pages 15–32. PLoPD4. Addison-Wesley, Massachusets, USA, 2000.

[8] Nicolás Cardozo and Siobhán Clarke. Context slices: Lightweight discovery of behavioral adaptations. In *Proceedings of the Context-Oriented Programming Workshop*, COP'15, pages 2:1–2:6. ACM, July 2015. doi: http://dx.doi.org/10.1145/2786545.2786554.

[9] Nicolás Cardozo, Sebastian Günther, Theo D'Hondt, and Kim Mens. Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations. In *Proceedings of the International Conference on Software Engineering Advances*, ICSEA'11, pages 130 – 135. IARIA, October 2011.

[10] Nicolás Cardozo, Sebastián González, and Kim Mens. Uniting global and local context behavior with context petri nets. In *Proceedings of the International Workshop on Context-Oriented Programming*, number 3 in COP'12, pages 1–3. ACM Press, 2012. doi: 10.1145/2307436.2307439. 11 June 2012. Co-located with ECOOP.

[11] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, and Theo D'Hondt. Modeling and analyzing self-adaptive systems with context petri nets. In *Proceedings of the Symposium on Theoretical Aspects of Software Engineering*, TASE'13, pages 191 – 198. IEEE Computer Society, July 2013.

[12] Pascal Costanza. Dynamically scoped functions as the essence of AOP. *ACM SIGPLAN Notices*, 38(8):29–36, 2003. ISSN 0362-1340. doi: 10.1145/944579.944587.

[13] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, October 2005. doi: 10.1145/1146841.1146842.

[14] Brecht Desmet, Jorge Vallejos, and Pascal Costanza. Introducing Mixin Layers to Support the Development of Context-Aware Systems. In *European Workshop on Aspects in Software*, EWAS'06, August 2006.

[15] Anind K. Dey. Understanding and using context. *Personal Ubiquitous Computing*, 5(1):4–7, January 2001.

[16] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006. ISSN 0164-0925. doi: 10.1145/1119479.1119483.

[17] Benoît Duhoux, Bruno Dumas, Hoo Sing Leung, and Kim Mens. Dynamic Visualisation of Features and Contexts for Context-Oriented Programmers. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS'19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367455.

[18] Benoît Duhoux, Kim Mens, and Bruno Dumas. Implementation of a feature-based context-oriented programming language. In *Proceedings of the Workshop on Context-oriented Programming*, COP'19, pages 9–16. ACM, 2019.

[19] Achiya Elyasaf and Arnon Sturm. Towards a framework for analyzing context-oriented programming languages. In *Proceedings of the 13th ACM International Workshop on Context-Oriented Programming and Advanced Modularity*, COP'21, page 16–23, New York, NY, USA, 2021. ACM. ISBN 9781450385428. doi:

10.1145/3464970.3468414.

[20] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, March 2006. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2006.61.

[21] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. Programming language support to context-aware adaptation: a case-study with Erlang. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'10, pages 59–68, New York, NY, USA, 2010. ACM Press. ISBN 978-1-60558-971-8. doi: 10.1145/1808984.1808991.

[22] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. 1983. ISBN 0-201-11371-6.

[23] Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the Dynamic Languages Symposium*, pages 77–88. ACM Press, October 2007. ISBN 978-1-59593-868-8. doi: 10.1145/1297081.1297094.

[24] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008. ISSN 0948-6968. doi: 10.3217/jucs-014-20-3307.

[25] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Proceedings of the International Conference on Software Language Engineering*, volume 6563, pages 246–265, 2011. ISBN 978-3-642-19439-9. doi: 10.1007/978-3-642-19440-5_15.

[26] Sebastián González, Kim Mens, Marius Colacioiu, and Walter Cazzola. Context Traits: dynamic behaviour adaptation through run-time trait recomposition. In *Proceedings of International Conference on Aspect-Oriented Software Development*, AOSD'13, pages 209–220, New York, NY, USA, 2013. ACM Press. ISBN 978-1-4503-1766-5. doi: 10.1145/2451436.2451461. URL `http://doi.acm.org/10.1145/2451436.2451461`.

[27] Sebastian Günther and Sagar Sunkle. rbFeatures: Feature-oriented programming with Ruby. *Science of Computer Programming*, 77(3):152–173, March 2012. ISSN 0167-6423. doi: 10.1016/j.scico.2010.12.007.

[28] Stephan Herrmann. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/-Java. *Applied Ontology*, 2(2):181–207, April 2007.

[29] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An Introduction to Context-Oriented Programming with ContextS. In *Generative and Transformational Techniques Generative and Transformational Techniques in Software Engineering*, GTTSE'07, pages 396 – 407, 2007.

[30] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March–April 2008. URL `http://www.jot.fm/issues/issue_2008_03/article4/`.

[31] IBM. An architectural blueprint for autonomic computing. Technical Report 3, IBM Research, June 2005.

[32] Hiroaki Inoue and Atsushi Igarashi. A Library-Based Approach to Context-Dependent Computation with Reactive Values: Suppressing Reactions of Context-Dependent Functions Using Dynamic Binding. In *Companion Proceedings of the International Conference on Modularity*, MODULARITY Companion'16, pages 50–54, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340335.

[33] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proceedings of International Conference on Aspect-Oriented Software Development*, AOSD'11, pages 253–264. ACM Press, March 2011.

[34] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A unified context activation mechanism. In *Proceedings of the 5th International Workshop on Context-Oriented Programming*, COP'13, pages 2:1–2:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2040-5.

[35] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *International Conference on Modularity*, MODULARITY'15. ACM, March 2015.

[36] Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara, and Tetsuo Tamai. Context-Oriented Software Development with Generalized Layer Activation Mechanism. *Transactions Modularity and Composition*, 1:3–40, 2016.

[37] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Push-based reactive layer activation in context-oriented programming. In *Proceedings of the 9th International Workshop on Context-Oriented Programming*, COP'17, pages 17–21, July 2017.

[38] Guillaume Kasin. Engineering context-oriented applications. Master's thesis, Université catholique de Louvain, June 2012.

[39] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. ISBN 0-262-11158-6 (hardcover), 0-262-61074-4 (paperback).

[40] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. volume 1241, pages 220–242. 1997. ISBN 978-3-540-63089-0. doi: 10.1007/BFb0053381.

[41] Paul Leger, Hidehiko Masuhara, and Ismael Figueroa. Interfaces for modular reasoning in context-oriented programming. In *International Workshop on Context-Oriented Programming and Advanced Modularity*, COP'20, pages 1–7, July 2020.

[42] Max Leuthäuser and Uwe Aßmann. Enabling view-based programming with scroll: Using roles and dynamic dispatch for establishing view-based programming. In *Proceedings of the Joint Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, MORSE/VAO '15, pages 25–33, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3614-7.

[43] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming*, 76(12):1194–1209, December 2011. ISSN 0167-6423. doi: 10.1016/j.scico.2010.11.013.

[44] Pattie Maes. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12):147–155, December 1987. ISSN 0362-1340. doi: 10.1145/38807.38821.

[45] Baptiste Maingret, Frédéric Le Mouël, Julien Ponge, Nicolas Stouls, Jian Cao, and Yannick Loiseau. Towards a decoupled context-oriented programming language for the internet of things. In *Proceedings of the 7th International Workshop on Context-Oriented Programming*, COP'15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3654-3.

[46] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. A context-oriented software architecture. In *Proceedings of the International Workshop on Context-Oriented Programming*, COP'16, pages 7–12. ACM, 2016.

[47] Vivek Nallur, Nicolás Cardozo, and Siobhán Clarke. Clonal plasticity: A method for decentralized adaptation in multi-agent systems. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'16, pages 122–128, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4187-5.

[48] Luís Pina and João P. Cachopo. Atomic dynamic upgrades using software transactional memory. In *International Workshop on Hot Topics in Software Upgrades*, HotSWUp'12, pages 21 – 25, Zurich, Switzerland, June 2012. IEEE.

[49] Thibault Poncelet and Loïc Vigneron. The Phenomenal gem: Putting features as a service on rails. Master's thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, June 2012.

[50] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 141–147. ACM Press, 2002. ISBN 1-58113-469-X. doi: 10.1145/508386.508404.

[51] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. JavAdaptor —flexible runtime updates of Java applications. *Software: Practice and Experience*, 2012. ISSN 1097-024X. doi: 10.1002/spe.2107.

[52] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009. ISSN 1556-4665. doi: 10.1145/1516533.1516538.

[53] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *Proceedings of International Conference on Aspect-Oriented Software Development*, AOSD'12, pages 191–202. ACM Press, 2012. ISBN 978-1-4503-1092-5. doi: 10.1145/2162049.2162072.

[54] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, August 2012. ISSN 0164-1212. doi: 10.1016/j.jss.2012.03.024.

[55] Lee Salzman and Jonathan Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. volume 3586, pages 312–336, 2005. ISBN 978-3-540-27992-1. doi: 10.1007/11531142_14.

[56] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. *Generative Programming and Component Engineering: Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003. Proceedings*, chapter A Selective, Just-in-Time Aspect Weaver, pages 189–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[57] Fabio Schreiber and Emanuele Panigati. Context-Aware Software Approaches: a Comparison and an Integration Proposal. In *Italian Symposium on Advance Database Systems*, SEBD'14, pages 175–184, 2014.

[58] Nigel Shadbolt. Ambient intelligence. *IEEE Intelligent Systems*, 18(4):2–3, 2003.

[59] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002.

[60] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM Press, 1984. ISBN 0-89791-125-3. doi: 10.1145/800017.800513.

[61] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. A dynamic instance binding mechanism supporting run-time variability of role-based software systems. In *Proceedings of the International Workshop on Live Adaptation of Software Systems*, LASSY'16. ACM, March 2016.

[62] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated generic functions: Enabling context-dependent method dispatch. In Benoît Baudry and Eric Wohlstadter, editors, *Software Composition*, volume 6144, pages 66–81, 2010. ISBN 978-3-642-14045-7. doi: 10.1007/978-3-642-14046-4_5.

[63] Tom Van Cutsem and Mark S. Miller. `traits.js`: Robust object composition and high-integrity objects for ECMAScript 5. In *Proceedings of the ACM International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC'11, pages 1–8. ACM Press, 2011. ISBN 978-1-4503-1171-7. doi: 10.1145/2093328.2093330.

[64] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. Hotwave: Creating adaptive tools with dynamic aspect-oriented programming in java. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE'09, pages 95–98, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-494-2.

[65] M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research at PARC in the late 1980s. *IBM Systems Journal*, 38(4):693–696, 1999. ISSN 0018-8670.

[66] Erwann Wernli, David Gurtner, and Oscar Nierstrasz. Using first-class contexts to realize dynamic software updates. In *Proceedings of the European Smalltalk User Group (ESUG'11)*, pages 21 – 31, 2011.