Context-Oriented Programming with the Ambient Object System

Sebastián González, **Kim Mens**, **Alfredo Cádiz**
Département d'ingénierie informatique
Université catholique de Louvain
`sebastian.gonzalez|kim.mens|alfredo.cadiz@uclouvain.be`

Abstract In this paper we present *AmOS*, the Ambient Object System that underlies the Ambience programming language. *AmOS* implements a computation model that supports highly dynamic behaviour adaptation to changing contexts. Apart from being purely object-based, *AmOS* features first-class closures, multimethods and contexts. Dynamic method scoping through a subjective dispatch mechanism is at the heart of our approach. These features make of *AmOS* a very simple and elegant paradigm for context-oriented programming.

Key Words: context-oriented programming, subjective dispatch, multiple dispatch, prototype-based programming, ambient intelligence
Category: D.3.3 [Programming Languages]: Language Constructs and Features

1 Introduction

In the vision of Ambient Intelligence [25], people are assisted in their everyday activities through the proactive, opportunistic support of non-intrusive computing devices offering intuitive interaction modalities. The usefulness and quality of delivered services could be improved considerably if devices were able to adapt their behaviour according to sensed changes in their surrounding environment, both at the physical and logical levels. This interplay between context-awareness and dynamic software adaptability is key to the construction of applications that are smart with respect to user needs. Unfortunately, current applications are hardly adaptable. Most applications exhibit fixed functionality and seldom do they sense their environment and adapt their services in a context-aware fashion. Many chances of delivering improved services to users and network peers are thus missed.

The need for adequate programming abstractions that enable application context-awareness has given rise to the emerging field of Context-Oriented Programming [17]. Our approach follows the same direction. This paper presents a programming model to ease the construction of applications that can react to changes in their execution context by adapting their behaviour dynamically. The starting point of our research is the development of novel language abstractions and the adaptation of existing abstractions that can render context-aware, self-adaptable applications easier to develop. We demonstrate that a simple yet powerful computation model based on classless objects and multimethods readily provides the needed support, leading to (a) straightforward application code that is not concerned with context adaptation, (b) behaviour that can be adapted dynamically to different contexts in a non-intrusive fashion (without modifying existing application code), and (c) context-aware applications with software architectures that are not biased towards context adaptation —rather, they can be designed freely according to their domain.

Whereas our model has been presented in the past using a Smalltalk-like surface syntax [15], its core has been written, and is therefore readily available, in Common Lisp. We call this core the Ambient Object System (*AmOS*). *AmOS* does not rely on CLOS,[1] in particular because *AmOS* is not based on the notion of *class*. In essence, *AmOS* is a prototype-based computation model [22] featuring multimethods and subjective dispatch [24]. In complement to a previous paper where we illustrated the main features of our Ambience language and how they support run-time adaptation of mobile applications to changing contexts [15], in this paper we open up the inner workings of the underlying object system (*AmOS*) and discuss its advantages for context-oriented programming.

To give the reader a first feel of the language before diving into the core abstractions of our model, the following section introduces a simple *AmOS* program that will serve as running example throughout the paper.

2 Motivating example

The example illustrates how to program the behaviour of a mobile phone and the way such behaviour can be adapted to context. We deliberately avoid a detailed explanation of the semantics behind the language constructs used in this example, relying on the reader's intuition instead. In the forthcoming sections we revisit this example as we gradually introduce the different language features in more detail. The example concentrates on functionality related to receiving and advertising calls on mobile phones, with the following basic requirements:

1. New incoming calls are advertised by playing a predefined ringtone.
2. Urgent calls are treated with priority over normal calls.
3. If the phone is off-hook (in use), a call waiting signal is played instead.

We divide the example in two parts. From a programmer's perspective, we show the code that needs to be developed and deployed on the mobile phone. From a user's perspective, we show the code that is executed and the resulting behaviour at run time, according to the context of use.

[1] CLOS is the standard object-oriented extension of Common Lisp.

2.1 Development Time

One of the key features of *AmOS* is the support of first-class contexts. Contexts are objects representing physical or logical properties of the environment in which the system is running. These properties may be about the user, the machine, the surroundings or in general any information which is computationally accessible [17], be it acquired through sensor input, network communication, generated internally, or otherwise. In our example, we first create a `@telephony` context, representing a prototypical situation in which a telephony service is available. Inside a mobile phone such service always is:[2]

```
(defcontext @telephony)
```

By convention, prototype names are prefixed with the `@` symbol. The `@telephony` context thus created is a plain object, without any special status in comparison to other objects in the system. Next we proceed to define objects and behaviour that are specific of telephony context. We thus request `@telephony` to be the currently active context:

```
(in-context @telephony)
```

All forthcoming definitions will belong to this context. Other existing contexts will remain unchanged.

For the sake of the example, a phone object contains a call manager and a speaker on which to advertise incoming calls:[3]

```
(defproto @phone (clone @object))
(add-slot @phone 'calls (clone @call-manager))
(add-slot @phone 'speaker (clone @phone-speaker))
(defproto @mobile-phone (extend @phone)))
```

As a result of `extend`, `@mobile-phone` will delegate to `@phone`.[4] All behaviour not understood directly by the former will be handed over to the latter. The call manager features four queues for call management:

```
(defproto @call-manager (clone @object))
(defproto @call-queue (extend @queue))
(add-slot @call-manager 'incoming (clone @call-queue))
(add-slot @call-manager 'ongoing (clone @call-queue))
(add-slot @call-manager 'terminated (clone @call-queue))
(add-slot @call-manager 'missed (clone @call-queue))
```

[2] Aimed at improving understandability, the `defcontext` construct is syntactic sugar for `(defslot @telephony (clone @context))`. This adds a slot named `@telephony` to the current context object whose value is a clone of the prototypical context object.

[3] The `defproto` construct is a synonym of `defslot` for addition of a slot with given name and value to the current context. We prefer the use of `defproto` over `defslot` because it encodes explicitly the programmer's intention.

[4] For a discussion of delegation in prototype-based languages and how it differs from class-based inheritance, see the seminal paper by Lieberman [20] and the book edited by Noble et al. [22].

The `@call-queue` prototype is a specialised form of queue for managing calls. The head of the `ongoing` queue (if present) is the currently active call; all other calls in the queue are on hold.

Still in telephony context, we define a phone call as an object that can be received on any phone:

```
(defproto @call (clone @object))
(defmethod receive ((call @call) (phone @phone))
  (advertise call phone)
  (enqueue call (incoming (calls phone))))
```

The `receive` multimethod is specialised on both `@call` and `@phone`. It encodes the prototypical behaviour for receiving calls on a phone: the call is advertised and added to the queue of incoming calls. The `advertise` method encodes the prototypical way of announcing a call to the user:

```
(defmethod advertise ((call @call) (phone @phone))
  (format t "Playing ringtone through ~a" (speaker phone)))
```

This tackles requirement 1 set forth previously.

AmOS methods, even when belonging to the same context, can be overloaded by using the same name but different specialisers. For example, behaviour that is better suited for urgent calls can be defined by overloading `enqueue` as follows:

```
(defproto @urgent-call (extend @call))
(defmethod enqueue ((call @urgent-call) (queue @call-queue))
  (push call queue))
```

This version of `enqueue`, specially conceived for urgent calls, puts the call in the front of the call queue instead of at the end. This tackles requirement 2.

As illustrated by the previous example, overloaded multimethods permit defining behaviour that is better suited to specific kinds of arguments. In addition to having this explicit dependency on their arguments kinds, *AmOS* methods have an implicit dependency on the context in which they are defined, and thus can be overloaded on that context as well, as shown next. Advertising behaviour that is specific to situations in which the phone is off-hook can be defined as follows:[5]

```
(defcontext @off-hook)
(with-context @off-hook
  (defmethod advertise ((call @call) (phone @phone))
    (format t "Playing call waiting signal through ~a~%"
              (speaker phone))))
```

This tackles requirement 3. The adapted behaviour for `@off-hook` context is specified in a non-intrusive way, leaving the original `advertise` method untouched.

[5] The `(with-context c b ...)` construct is syntactic sugar for the following: `(activate c) b ... (deactivate c)`; `with-context` makes sure that `c` is deactivated at the end even if control flow ends prematurely in `b` because of (e.g.) an exception. Context activation is explained in Section 5.3.

Encoding behaviour in this context-oriented way is therefore fundamentally different from using conditional statements.

To complete the example, we still have to show the way `@off-hook` is managed. The context is activated when a call is answered:

```
(defmethod answer ((phone @phone))
  (let ((call (dequeue (incoming (calls phone)))))
    (push call (ongoing (calls phone))))
  (activate @off-hook))
```

Correspondingly, `@off-hook` is deactivated upon hang up:

```
(defmethod hang-up ((phone @phone))
  (deactivate @off-hook)
  (let ((call (pop (ongoing (calls phone)))))
    (enqueue call (terminated (calls phone)))))
```

All code shown so far is written at development time and deployed into the phone.

2.2 Run Time

During normal use, actual mobile phones and phone calls are created by cloning respective prototypes:

```
(defslot bobs-phone (clone @mobile-phone))
(defslot alices-call (clone @urgent-call))
```

Behaviour is triggered by invoking multimethods like `receive`. The default output is:

```
(receive alices-call bobs-phone) →
  Playing ringtone through phone speaker
```

The output of the *same* expression is different when the phone is in use:

```
(receive alices-call bobs-phone) →
  Playing call waiting signal through phone speaker
```

More advanced examples of context adaptation will be shown after having explained the basic mechanisms that underly our approach.

3 *AmOS* Core Concepts

AmOS aims at being a multiparadigm model that does not sacrifice simplicity and homogeneity for expressiveness and flexibility. Section 2 gave a first glimpse of that from an end-user perspective. In the remainder of the paper we show that simplicity and homogeneity are at the core semantics of the object model. We start by highlighting the underlying concepts that have been introduced in an intuitive fashion so far. These concepts form the cornerstones of the object model, on which all the rest is based.

Objects Every first-class entity in *AmOS* is an object — that is, the model is *purely* object-based. The observable properties of objects are their identity, acquaintances and behaviour. Whereas identity is an immutable (defining) characteristic, acquaintances and behaviour can vary over time. The latter two thus constitute the *state* of an object.

AmOS objects are said to be *open*, since new methods and attributes can be added or removed at run time, without editing previously existing code. Open objects are analogous to *open classes* [4] in class-based languages.

Some objects in the system act as representative examples of domain entities, and are therefore called *prototypes*. However, prototypes do not have a special status in the language other than being meaningful exemplars [20, 22].

Cloning New objects can be created by cloning existing ones. Cloned objects have a distinct, unique identity, but are identical to the cloned object in all other regards.

Messages Interaction among objects happens through message passing. A message is a request for interaction among the participants involved in the message. To this effect, each message has a *selector* object that identifies the desired interaction, and an argument list of objects that will take part in it. Messages are *symmetric*: there is no distinguished receiver for any given message.

Delegation Behaviour can be delegated from one object to another by placing a delegation link between them. When we refer to *inheritance* in this paper we mean such delegation-based inheritance. Since objects can have multiple delegations, a directed graph of delegation links can be formed. Messages that are not understood by an object can be handled by one of the delegates in the delegation graph. Cyclic delegations are supported, as explained in Section 4.2. Sample delegations are shown in Figure 1.

Methods Methods describe prototypical interactions among objects. Every method has a selector that identifies the particular interaction it implements, and a list of prototypical objects that take part in the interaction. The method is said to be *specialised* on those particular objects.

Rather than belonging to a single class as in Java or to a single generic function as in CLOS, *AmOS* methods belong simultaneously to all their specialisers. In other words, method ownership is shared, both at a conceptual and technical levels. Methods are thus *symmetric*, just like messages are.

Because of shared ownership, a method can be accessed only if the client holds references to suitable arguments and suitable contexts to which the

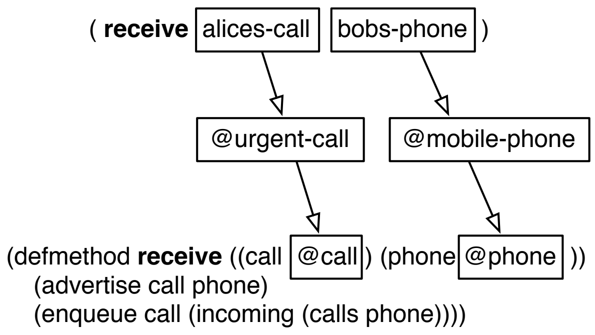
(receive alices-call bobs-phone)
@urgent-call
@mobile-phone
(defmethod receive ((call @call) (phone @phone))
(advertise call phone)
(enqueue call (incoming (calls phone))))

Figure 1: Method applicability for a given message. The hollow-headed arrows denote delegation relationships.

method is applicable. This brings in the advantages of capability-based security to the model [21]. In contrast, generic functions in CLOS are globally visible objects conferring centralised access to all homonym methods.

Method applicability For any given message, a method is *applicable* if the selector and arguments of the message match those of the method. The selectors match if they have the same object identity.[6] The arguments match if each message argument delegates in zero or more steps to the method specialiser in the same position, as illustrated in Figure 1.

Method specificity Due to multiple inheritance, more than one method might be applicable for any given message. A notion of *specificity* is introduced to solve ambiguities, which is a strict, total order relationship among methods. A second source of ambiguity is multiple dispatch. To solve this kind of ambiguity, *asymmetric dispatch* [4] is used, giving earlier message arguments more importance during dispatch than later arguments. With these rules there will always be a method that is more specific than the others and can therefore be chosen for execution.

These concepts are all there is to the basic computation model of *AmOS*. Perhaps the least trivial part is message disambiguation. This topic is discussed in Section 4.2. The next sections progressively show how the core concepts just explained are sufficient to support the fundamental constructs of our model, which in the end enable dynamic behaviour adaptation to context.

[6] Given that the only relevant property of a selector is its identity, any object can be used as selector, although most often symbols are used.

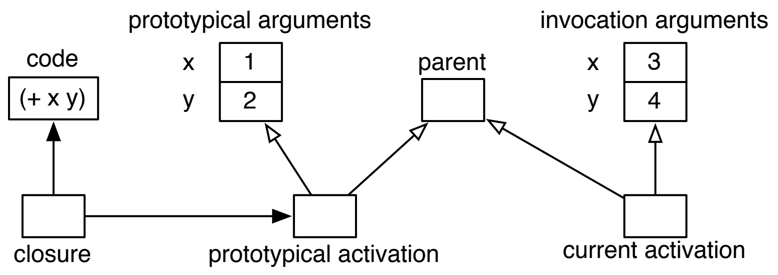

Figure 2: Prototypical activation and cloned activation with actual arguments. Solid arrows represent object references, the hollow arrows represent delegations.

4 Opening Up *AmOS*

The core concepts explained in Section 3 are not only meant for end programmers. The mechanisms used in the lowest levels are the same, namely objects, cloning, messages, delegation, and multimethods. In this section we describe the fundamentals of *AmOS* from a language engineer's perspective. This section shows that *AmOS* is an open system [19] and that such openness does not entail additional complexity at the conceptual and technical levels.

4.1 Closures and Activations

The most basic executable entity in *AmOS* is the *closure*. It has `lambda`-like syntax and semantics, as the following example illustrates:

```
(& (x y) (+ x y))  →  closure
```

Every closure has an associated *activation record* —hereafter simply called *activation*— which holds the dynamic information that is associated with its invocation. Activations are the environments in which closure code is executed.[7] Like in Self [3], activations are first-class objects.

It is possible to specify prototypical argument values to be held in the activation of a closure. They are placed next to each argument name:

```
(& ((x 1) (y 2)) (+ x y))  →  closure
```

This closure is illustrated in Figure 2. As can be seen, the prototypical activation delegates to an *arguments* object, which holds one slot per closure argument. Upon invocation, the closure activation is cloned and the prototypical arguments are substituted by the actual arguments. The closure's code is then executed in this freshly created environment and is thus fully reentrant. Figure 2 shows the fresh activation resulting from the following invocation:

[7] Activations are the object-based version of what is usually known as *stack frames* in other models.

```
(invoke (& ((x 1) (y 2)) (+ x y)) (list 3 4))  →  7
```

Each activation delegates to a *parent* object, also illustrated in Figure 2. Messages not understood by the current activation or by its arguments object are delegated to the parent.[8] The parent corresponds to the enclosing lexical scope of the closure, so that outer definitions can be seen inside the closure's environment. For the particular case of the *top-level activation*, which has no enclosing lexical environment, the parent is the so-called *current context*. This context link is crucial to our approach and is explained further in Section 5.

As shown in this section, the semantics of closures involves nothing more than objects, cloning and delegation. The next section explains methods and their dispatch infrastructure.

4.2 Methods and Specialisation

Methods are obtained by enriching closures with a dispatch mechanism. Since methods are extended forms of closures, the execution semantics described in Section 4.1 applies unmodified to methods. In the case of methods, the prototypical arguments are considered to be *argument specialisers*. The code of the method is designed to work for those specialisers in particular, and for any extension (through delegation) thereof. Reconsider for instance the `receive` method:

```
(defmethod receive ((call @call) (phone @phone))
  (advertise call phone)
  (enqueue call (incoming (calls phone))))
```

The `receive` method is basically a named closure with prototypical arguments `@call` and `@phone`, which are used as specialisers.

Roles

The link between a method and its specialisers is established through *roles*, originally proposed in the Prototypes with Multiple Dispatch model [24]. Any object that is used as method specialiser plays a role in the interaction described by the method. As illustrated in Figure 3, the argument specialisers `@call` and `@phone` play a role in the `receive` interaction, at the first and second positions respectively. The illustrated roles are triplets (s, i, m) of the selector s identifying the interaction, the position i at which the object plays the role, and the method m implementing the behaviour.

Figure 3 also shows the conceptual difference among the different kinds of objects. Objects in the *plain* layer correspond to concrete domain entities that are being manipulated at the moment; objects in the *prototypes* layer are prototypical (usually meant for cloning, rather than direct manipulation); finally, the

[8] The order of delegations is important here. The arguments have more precedence than the parent by having an earlier position in the delegation list of the activation. Figure 2 does not depict this order.

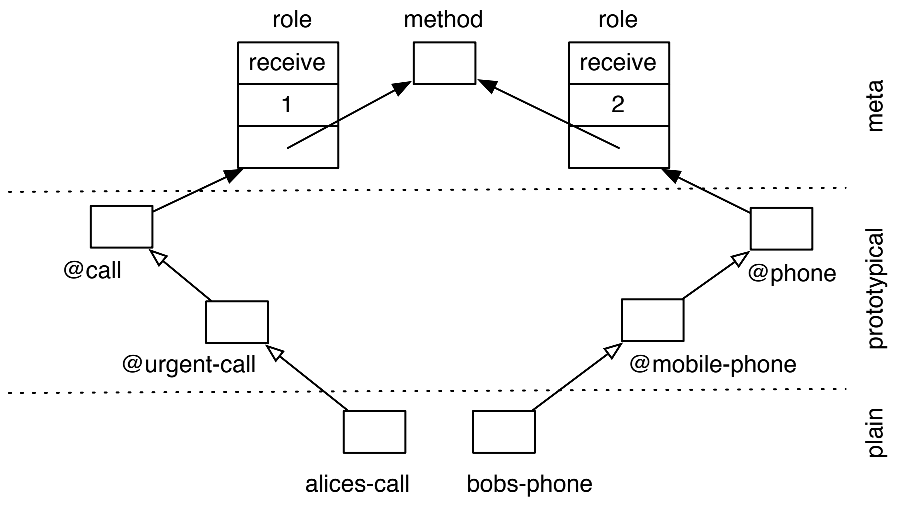

Figure 3: Roles corresponding to the `receive` method specialised on the `@call` and `@phone` prototypes, and arguments `alices-call` and `bobs-phone` for which the method is applicable.

core computation model is available through a series of *meta* objects describing base objects, their roles, methods, and so on.

Method selection

Method overloading brings about the problem of choosing the method version that is best suited to the given arguments. Specificity among applicable methods is defined by *rank vectors* [24]. Each rank vector entry contains the delegation distance between the message argument and corresponding method specialiser. For instance, the rank vector of the method illustrated in Figure 3 for the message with arguments `alices-call` and `bobs-phone` is $(2, 2)$, since the path in the delegation graph that goes from message argument to method specialiser is of length 2 for both arguments. A rank vector with only zeroes is a "perfect match", corresponding to the case where the message arguments are the very method specialisers.

We use an adapted version of the C3 linearisation algorithm [1] to topologically sort the delegation graph of each message argument and have a well-defined notion of distance. Our adaptation of C3 supports delegation cycles trivially, by taking into account only the first occurrence of a delegate in the linearisation and ignoring any further occurrences arising from cycles. Next to our handling of cycles, we also need an automatic resolution strategy for inconsistent delegation graphs (that cannot be linearised by C3). Such automatic strategy is necessary in *AmOS*, as ambiguities cannot always be detected at development time due to dynamic inheritance. Delegation graphs can change arbitrarily at run time, opening the door for ambiguous cases that could be precluded in systems with static inheritance such as Cecil [2].

Ambiguities arising from multiple dispatch —for example, considering whether the rank vector $(1, 2)$ is more specific than $(2, 1)$— are resolved by imposing left to right argument precedence as in CLOS (i.e. a lexicographic ordering): $(1, 2)$ is thus considered more specific than $(2, 1)$. As a consequence, methods with a better match in earlier argument positions will be considered more specific than other applicable methods. This choice is justified by observing that, in practice, important arguments tend to have earlier argument positions, whilst more auxiliary arguments are usually placed rightwards; the extreme case is observed in languages with single dispatch, in which only the leftmost argument is dispatched dynamically and therefore completely determines selected behaviour.

Method specialisation is useful in defining behaviour for special kinds of objects and dealing with particular cases without hard-coding conditional statements [13]. Section 5 explains the way we further exploit specialisation and multiple dispatch to define *context-specific* behaviour, and the way such behaviour can be adapted dynamically as needed. Before proceeding, we explain one last important element of the computation model, accessor methods.

4.3 Accessor Methods

Sticking to a *pure* object-based semantics, in *AmOS* there is no such thing as a variable access. Everything in *AmOS* is done through message passing. In particular, argument accesses are actually method invocations (as in Self [29]), even though on the surface they look like plain variable accesses. In spite of the fact that variable accesses do not take any explicit arguments, it should be taken into account that there is always an implicit argument (the current activation) which is passed with every method invocation, as explained in Section 5.

To illustrate accessor methods, we revisit the `receive` method once more:

```
(defmethod receive ((call @call) (phone @phone))
  (advertise call phone)
  (enqueue call (incoming (calls phone))))
```

In the body of the method, the occurrences of the symbols `call` and `phone`, which are the parameter names, are replaced by message sends.[9] The symbols `@call` and `@phone` are also messages under the surface.[10] The code just shown is equivalent to the following:

[9] Thanks to Common Lisp's `symbol-macrolet` facility.
[10] They are defined with Common Lisp's `define-symbol-macro` facility.

```
(defmethod receive ((call (@call)) (phone (@phone)))
  (advertise (call) (phone))
  (enqueue (call) (incoming (calls (phone)))))
```

In this code it is apparent that everything is done through message passing. The choice between the first (implicit) syntax and second (explicit) syntax for accessing arguments is left to users.

We call the accessors discussed so far *inner accessors*, because they are used to access the slots of an object "from the inside" —that is, when the object is being used as an evaluation environment, as activations for example are normally used. Accesses to arguments in activations are not the only uses of inner accessors. Prototypes, usually stored in context objects, are also accessed by means of inner accessors. For example, the `(@call)` and `(@phone)` messages invoke inner accessors that read slots from `@telephony` context.

Besides inner accessors, *AmOS* features other kinds of accessors. In particular, *outer accessors* read slots form the "outside" of an object, as the accessor `age` in the following expression does: `(age person)`. In this example, the outer accessor receives an explicit argument `person` from which it is supposed to read a slot —or write it, as in the expression `(setf (age person) 31)`.

Accessor methods have no special status and use no special semantics to access the slots of objects for which they have been defined. Given that accessors are normal methods, it is possible to define context-specific accessors, and hence to have slots whose apparent value depends on the context from which they are observed.

5 Context-Oriented Programming in *AmOS*

Context-Oriented Programming enables the expression of behavioural variation according to context [17]. Dynamically adaptable context-aware applications can be written elegantly[11] thanks to specific linguistic support to deal with behavioural context dependencies. As illustrated in Section 2, *AmOS* provides dedicated language abstractions such as `defcontext` and `with-context` to encode context-dependent behaviour. This section explains the foundations of those abstractions.

5.1 Main Cornerstone: Dynamic Scoping of Behaviour

Run-time behaviour adaptation is supported in *AmOS* by introducing a kind of dynamic scoping mechanism for methods. Generally speaking, the main reason why dynamic scoping is useful is that it allows the caller's state to influence the behaviour exhibited by the callee in a deep fashion (i.e. across nested method

[11] Read: concisely, legibly, with simplified control flow and with little or no tangling and scattering of source code.

calls). Such influence is not intertwined in the form of arguments that must be passed from one method to the next. Clearly, having such pass-through arguments is quite inconvenient, as the arguments crosscut all methods and messages that need to be influenced [5], and all possible influences that might prove useful must be foreseen and hard-coded in method signatures. Dynamic scoping can help alleviating these problems.

Many languages that support dynamic scoping, such as Common Lisp and some dialects of Scheme, have an intrinsic concept of *variable*. These languages must draw a distinction between dynamic scoping of variables and functions. Given that the concept of variable is not intrinsic to *AmOS* (as explained in Section 4.3), we need be concerned only with dynamic scoping of methods in our discussion.

5.2 Dynamic Scoping in an Object-Based World

AmOS identifies dynamic scoping —a concept coming mainly from the functional programming world— with subjective behaviour —a concept coming from the object-oriented world [27]. Subjective behaviour is roughly equivalent to dynamic scoping: it is behaviour that depends on the caller's point of view or state.

The power of dynamic scoping, or similarly, of subjective behaviour, can be brought to the object-oriented world fairly easily under certain conditions. Any language with multiple dispatch can support subjective behaviour [27], merely by passing with every message an implicit argument that represents the current point of view or state of the caller. This implicit argument participates in the dispatch process as any other argument does. As a result, chosen behaviour will depend on this implicit subjective element [24].

In *AmOS*, the current activation of the executing closure or method is passed implicitly as first argument of every message. This way, behaviour selection will depend on the current execution environment of the sender. This simple exploitation of multiple dispatch results in a kind of dynamic scoping mechanism that is surprisingly convenient, as the remainder of the paper illustrates.

5.3 Context as a Graph of Delegating Objects

For any given message, applicable methods are first looked up in the current activation, and by following the lexical parent link, they are looked up further in enclosing lexical scopes, until the top-level activation is reached. Rather than stopping at this point by having an empty object be the parent of the top-level activation, we assign an object which we consider the *current context*. The current context can delegate further to other context objects as needed.

Figure 4 shows a sample configuration of activations and context objects corresponding to the invocation of the `receive` method. Activation parent links

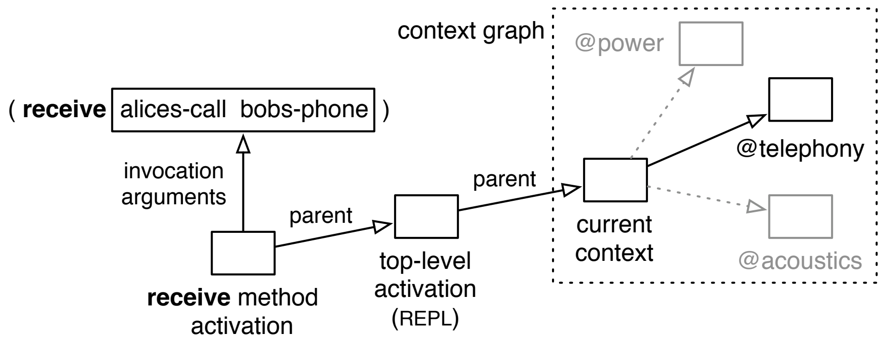
context graph
@power
(receive alices-call bobs-phone)
@telephony
invocation arguments
parent
parent
current context
@acoustics
receive method activation
top-level activation (REPL)

Figure 4: Invocation of the `receive` method. The lexical scope delegation chain is static, whereas the context graph topology is managed dynamically.

delegate to enclosing lexical scopes and are kept constant, in correspondence to the program text structure. Delegation links starting from the current context object and beyond are dynamically managed and may change at run time. Following normal delegation semantics, messages that are not understood by the static activation chain will be delegated to the current context.

The objects that are reachable by delegation starting from the current context constitute the *current context graph* (shown in the dashed box of Figure 4). A context that makes part of the current context graph is said to be *active*. The current context thus serves as an entry point to all currently active subcontexts. The reciprocal of the active status is of course *inactive*; any context that is not linked through delegation to the current context graph is inactive.

The context graph can be seen as a reification of the physical and logical environment in which the system is currently running. Each individual context object represents one part of such environment, and contains domain-specific information. For instance, the `@telephony` context has telephony-specific prototypes such as `@phone` and `@call`, method definitions such as `receive` and `advertise`, and contexts such as `@off-hook`.

5.4 Dynamic Adaptation through Context Manipulation

As explained previously, most messages are delegated to the current context. Hence, the current context graph plays a primary role in determining system behaviour. By manipulating context objects and their delegation relationships adequately, system behaviour can be adapted to the environment on the fly, as changes are detected.

In our example, the `answer` and `hang-up` methods are in charge of activating and deactivating the `@off-hook` context. When activated, this context is linked through delegation to the current context graph, and it is unlinked when deactivated; `@off-hook` is an example of a *transient* context. We think of contexts that have a more permanent nature, like `@telephony`, as *features*.[12] Having telephony support is a feature of a phone —indeed a very inherent one. Another example is an `@acoustics` feature which renders the phone aware of its acoustic environment, by including contexts such as `@silent`:

```
(defcontext @acoustics)
(in-context @acoustics)
(defcontext @silent)
```

The `@silent` context is part of `@acoustics`, much like `@off-hook` is part of `@telephony`. The `@silent` context can be activated when the system detects that (e.g.) a library, museum or hospital has been entered.

5.5 Discussion

We have shown at this point the most important elements of a computation model that is particularly well suited to context-oriented programming. The proposed representation of context as a graph of delegating objects has a number of advantages. Firstly, such representation is simple and concrete. This helps creating a sense of *tangibility* and *malleability* [26] of context. By exposing the representation to the programmer, it becomes possible to have a direct mental picture, and a clear programmatic understanding of what context is and how to manipulate it. Secondly, the connection between context and behaviour is immediate, making it easy to understand how context affects behaviour. Causality between context and behaviour comes as a natural consequence of regarding the context as an object (graph).[13] Thirdly, idiosyncratic contexts are supported naturally. Our approach naturally (paradigmatically) supports behaviour that is adapted to very specific contexts, such as one particular room of a building. Thanks to delegation, idiosyncratic contexts can exhibit more general behaviour as well.

6 Working with Contexts in *AmOS*

Having explained the foundations of context orientation in *AmOS*, we proceed to show a number of techniques to manage the changing context graph coherently.

[12] Contrast `@telephony` with `@off-hook`: it is unintuitive to think of `@off-hook` as a "feature".

[13] In a black-box view, the context is simply an object with behaviour, irrespective of whether this behaviour comes from delegation or not; in an open view of context, context structure is revealed and it becomes apparent that the context is actually a graph of delegating objects.

The more advanced context-oriented capabilities are illustrated by extending the running example introduced in Section 2.

6.1 Bypassing contexts

Suppose we want to add a *Discretion* extension to the phone. This extension includes call advertising behaviour that is better adapted to silent environments:

```
(with-context @silent
  (defmethod advertise ((call @call) (phone @phone))
    (format t "Activating phone vibrator~%")))
```

This new version of `advertise` activates the phone vibrator, without producing sound. The *Discretion* extension thus makes the phone more adaptable to silent environments:

```
(activate @silent) →
  Switching @silent on

(receive alices-call bobs-phone) →
  Activating phone vibrator
```

Now incoming calls activate the phone vibrator instead of playing the ringtone when the phone is in a silent context. If the silent context is deactivated, behaviour reverts to the default playing of a ringtone.

When the phone is off-hook (i.e. the user is talking) and a new incoming call is detected, the phone should not vibrate, even if running in `@silent` context. It would feel bizarre to suddenly receive physical vibration on the ear while talking with someone. To account for such situations, a specialised version of `advertise` can be defined as follows:

```
(with-context (@silent @off-hook)
  (defmethod advertise ((call @call) (phone @phone))
    (without-context @silent
      (resend))))
```

The `without-context` call executes the contained body in a context in which `@silent` is inactive; the phone will therefore not vibrate, as intended. Although this implementation appears to be sufficient, it has a problem that could become apparent in some situations. The `resend` call is made with an inactive `@silent` context. This means that whatever behaviour is eventually chosen by `resend` will *not* be adapted to silent environments, but rather be meant for default acoustics. Conceptually, it is wrong to disable the `silent` context in this rather drastic way, given that the surrounding environment has not actually changed — there is a potential mismatch between the outside world and the logic encoded in the method. The programmer's intention is simply to pick the next most specific behaviour that is not meant for silent environments, but such behaviour should be executed in a context that faithfully reifies the current environment. To remedy this situation, we introduce the `resend-bypassing-contexts` construct:

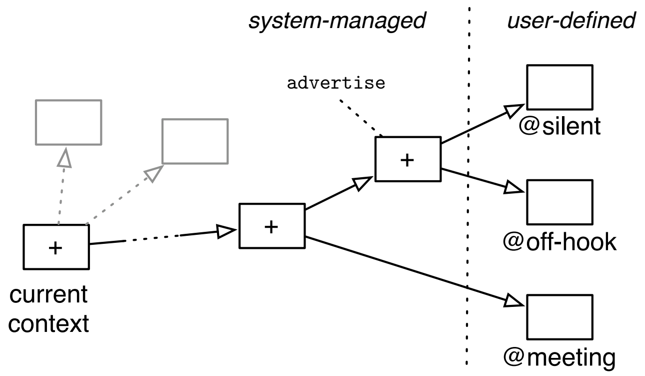

Figure 5: Context graph as combination of contexts.

```
(with-context (@silent @off-hook)
  (defmethod advertise ((call @call) (phone @phone))
    (resend-bypassing-contexts (list @silent))))
```

The `resend-bypassing-contexts` construct is a sort of context-oriented `super` call. When `@silent` and `@off-hook` are active simultaneously, the *Discretion* extension will give priority to off-hook behaviour over silent behaviour, but this will not entail the execution of the next most specific method in a wrong context.

6.2 Context combinations

A second feature of *AmOS* illustrated by the example above is *context combinations*. Note that the last version of `advertise` introduced in Section 6.1 is specialised on two context objects at the same time, namely `@silent` and `@off-hook`. When a list of contexts is passed to `with-context`, *AmOS* will make a context combination. Context combinations are context objects of their own, representing the combination as a whole.[14] Behaviour that is specific to the particular combination can be defined as exemplified by the version of `advertise` shown previously. Other behaviour not specific to the combination is delegated to the constituent subcontexts `@silent` and `@off-hook`, thanks to suitable delegation links illustrated in Figure 5. Contexts to the left-hand side of the figure are system-managed combinations (denoted by a plus `+` sign), whereas contexts on the right-hand side are user-defined contexts which can be seen as combinations of only one constituent context —the context itself. The run-time system is in charge of placing delegation links from more specific combinations to less spe-

[14] In a class-based model, a new class would need to be artificially introduced, of which the combination would be the sole instance. Our prototype-based model is free of such artefacts, naturally supporting singleton objects.

cific ones. The current context[15] is the most specific active combination at any given time. User-defined contexts are the least specific combinations there can possibly be. If needed, user-defined contexts can have delegation links to other subcontexts. Such delegations are not under control of the run-time system, and can be managed freely by the user.

At all times, there is at most one context object representing the combination of a given set of component subcontexts. For instance, the combination of `@silent` and `@off-hook` always results in the *same* combined context object that delegates to `@silent` and to `@off-hook`. If it were not the case, that is, if a new context object delegating to `@silent` and to `@off-hook` were created each time it were needed, then the methods that were specialised on the first version of the context combination would not be visible (applicable) to the second or any new subsequent versions that would be created, despite the fact that, conceptually, they represent the same combination. Conceptually there is only one `( @silent @off-hook )` combination, and computationally this must also be the case. On a practical level, this uniqueness property implies that created combination objects need to be stored by the context management system of *AmOS*, so that these same objects can be retrieved when required. For each combination request, the order of the given subcontexts is irrelevant.

6.3 Feature Interaction

Rather than introducing a new language feature, this section shows an example of behaviour interaction between base code and extension code. This makes part of our experience in working with contexts.

When people participate in certain activities, like a meeting, they should not be disturbed by their mobile phones. A *Call Forwarding* feature can understand the current situation and forward incoming calls to another predefined number during periods in which the user cannot be interrupted. We thus introduce a context representing a meeting situation:

```
(defcontext @meeting)
(add-delegation @meeting @silent)
```

Since typical meeting situations are supposed to be silent, we explicitly a delegation link from `@meeting` context to `@silent` context. When the system detects a meeting situation, it will activate this prototypical context:

```
(activate-context @meeting)  →
  Switching @silent on
  Switching @meeting on
```

As can be observed, activating the `@meeting` context implies activating related contexts as well, due to delegation relationships. Hence, behaviour that is adapted to silent environments will be active during meetings as well.

[15] Recall the dashed box of Figure 4.

Off-hook	Silent	Meeting	Forwarding	Behaviour
×	×	×		Ringtone
×	✓	×		Vibrator
×	✓	✓	×	Vibrator
×	✓	✓	✓	Call forwarding
✓	×	×		Call waiting signal
✓	✓	×		Call waiting signal
✓	✓	✓	×	Call waiting signal
✓	✓	✓	✓	Call forwarding

Table 1: Call receiving behaviour acording to context combinations. The "Forwarding" column represents the `forward-number` setting of the phone, rather than a context activation state.

The *Call Forwarding* extension adapts the default call reception behaviour of the phone as follows:

```
(add-slot @phone 'forward-number nil)
(with-context @meeting
  (defmethod receive ((call @call) (phone @phone))
    (if (forward-number phone)
        (format t "Forwarding ~a to ~a~%"
                  call (forward-number phone))
        (resend))))
```

Call Forwarding specialises `receive`; if the forwarding number is set, the call will be forwarded to that number, and the `advertise` method will not be invoked. On the other hand, if a forwarding preference has not been set (i.e. if it is `nil`), the `resend` call[16] will invoke the original behaviour as if the extension did not exist.

The *Discretion* feature (introduced in Section 6.1) and the *Call Forwarding* feature are deployed as separate modules that can be installed at will by the user. These extensions are independent, meaning that they do not need each other to work correctly: none, one or the two of them can be installed at any given time on the phone. Nonetheless, independence does not mean lack of interaction. The extensions do interact if both are installed on the same system, as can be observed in Table 1. The table illustrates the interactions of `@off-hook`, `@silent`, `@meeting` and the forward number setting. Not all 16 boolean combinations are interesting or even possible, and have thus been omitted from the table. In particular, the combinations where `@meeting` is active and `@silent` is inactive are impossible, because the activation of `@meeting` implies the activation of `@silent` by way of delegation. Further, the forwarding slot is unimportant when the `@meeting` context is inactive. The interactions are thus reduced to eight pos-

[16] The `resend` method is similar to `call-next-method` in CLOS.

sible and relevant cases, with four associated behaviours that can be exhibited by the phone.

Even though in this example the behaviour arising from feature interaction is appropriate or "wanted", this might not necessarily be the case in more complex situations. In a system with dozens or hundreds of features, "unwanted" interactions among features can arise [31]. We still need to devise a systematic way of handling feature interactions, as is done for instance in Prehofer [23].

7 Discussion

AmOS is a very dynamic computation model. It features dynamic dispatch,[17] dynamic inheritance, dynamic typing, and dynamic method scoping. One might very well wonder if such level of dynamism remains manageable. Although the answer is affirmative for small-scale scenarios, we still need to gather experience with larger case studies to assess the usefulness of the model in complex systems.

Open objects and multimethods allow clean separation of concerns [4]. Code corresponding to different concerns can be modularised in different methods, and new concerns can be added in the form of new methods, without modifying existing code. In particular, this good modularisation property helps separating context-dependent behaviour from base application behaviour cleanly.

We make a distinction between the *intrinsic* and *extrinsic* properties of objects [16]. The coordinates detected by a GPS for instance are intrinsic to the operation of the GPS —they are its *raison d'être*. The market price of the device, on the contrary, is an extrinsic property that might be interesting only from the point of view of a reseller. Similarly, the `@off-hook` context introduced in Section 2.1 is intrinsic to telephony, whereas the `@silent` context is extrinsic to telephony (it is intrinsic to acoustics). Hence, even though there is context-management code in the implementation of `hang-up` (defined in Section 2), we do not regard this as tangled code, because it is an integral part of the application's base logic. The exploitation of context-oriented programming for adaptation to intrinsic modes or states renders such base application logic adaptable to context, as illustrated by the running example.

Having many small behavioural pieces (i.e. multimethods) that might be applicable for any requested interaction (i.e. messages), behaviour composition becomes an issue. Flexible method combination techniques are necessary to deal with all behaviour that is applicable for a given message. *AmOS* does not yet incorporate advanced method combination techniques as those of CLOS for example, or as suggested by Harrison and Ossher [16]. In *AmOS*, all applicable methods are linearised, and more specific methods can decide at their discretion to invoke less specific methods by means of constructs such as `resend`

[17] This synonym of multiple dispatch emphasises the fact that behaviour selection depends on the *dynamic* value of *all* arguments, rather than only one or none.

and `resend-bypassing-contexts`.[18] The downside is that automatic method linearisation does not necessarily yield the "fittest" order in which to execute applicable behaviour [28]. More declarative and intentional approaches such as *predicate* dispatch [12] and *filtered dispatch* [6] could be used instead of, or in complement to, automatic linearisation.

We have not made performance measurements yet. However, given that message sends are fully reflective,[19] and there is no caching mechanism in place yet, chances are that our current implementation of *AmOS* does not match the speed of mature CLOS implementations and of CLOS extensions such as ContextL. Performance was, however, not our main concern in developing *AmOS*; our main focus was on language design.

Table 2 summarises the language features described in this paper and their associated advantages. In the table, *software feature* refers to bundles of attributes and methods belonging to (i.e. specialised on) a particular context, such as `@telephony`. The advantages marked with an asterisk are made possible by our approach, but proper support requires further refinement of our techniques. In particular, dynamic composition of features is currently limited by the linearisation semantics of method specificity mentioned previously. Regarding dynamic software feature activation and deactivation, we still need to provide adequate support to prevent the concurrent deactivation of a context that is being used. This problem is discussed further in González et al. [15].

8 Related Work

AmOS was initially inspired on Self [29] and Cecil [2], but later on adopted the similar, albeit more flexible, Prototypes with Multiple Dispatch (PMD) model [24]. Although the authors of PMD are well aware of the potential of subjective dispatch, they never fully exploited its possibilities. Subjectivity was left aside as happened with the Self extension Us [27]. We know of only one example showing the potential of subjective dispatch in the PMD model.[20] *AmOS* can be seen as a version of PMD that boosts subjective dispatch, making it as fundamental to the model as prototypes and multimethods.

Soon after adopting the PMD model we became aware of ContextL [7], a class-based cousin of *AmOS*, which also exploits a dynamic scoping mechanism to achieve behaviour adaptation. ContextL —an extension of CLOS— not only shares the similar goal of having behaviour depend on context, but also

[18] Another similar construct, `resend-as`, is described in González [14].

[19] This means that the `(send selector arguments)` meta-method is executed for every message, bringing in the advantages of meta-programming in our exploration of language semantics, to the detriment of performance.

[20] This example is shown in Salzman and Aldrich [24]. The distribution of the Slate language —PMD's reference implementation— contains no examples as of date, and the subjective dispatch feature is currently disabled in the interpreter.

Prototypes	Multimethods	Open objects	Multiple inheritance	Multiple dispatch	Dynamic inheritance	Dynamic scoping	Dynamic typing	Reified context	Context combinations	Bypassing resends	
•											Idiosyncratic contexts
	•	•									Separate software features
	•	•					•				Unanticipated new features
			•		•	•			•	•	Dynamic composition of features*
				•	•	•					Dynamic behaviour adaptation
					•	•					Dynamic software feature (de)activation*
•								•	•	•	Context-oriented composition

Table 2: Summary of language features (vertical) and their related advantages (horizontal).

a similar approach, by using an implicit argument that influences method dispatch. The analogous of *AmOS* contexts are ContextL *layers*. The differences between these two constructs are mostly idiomatic. Whereas *AmOS* considers context objects as direct reifiers of situations (being in a quiet environment, running with low battery power, etc.), ContextL does not propose such a direct semantic link between detected contexts and layers —in principle, layers are not seen as reifiers of anything in particular. Some other differences are however more fundamental. In ContextL there is one *layer configuration* (analogous to the context graph of *AmOS*) per thread. Threads cannot modify each other's layer configurations. Whereas thread locality ensures non-interference with other threads, such interference is sometimes useful. In *AmOS*, there is a unique context graph that is shared by all threads. *AmOS* implements immediate reaction to changes in context, whereas ContextL sticks to an initial context while finishing an ongoing computation. In *AmOS* the concurrent modification of the shared context graph can give rise to inconsistent behaviour [15]. In ContextL, the layer configuration follows a stack discipline. Once invoked, the behaviour of a running method cannot be influenced, unless context-switching constructs like `with-active-layers` and `ensure-active-layer` are used explicitly.[21] Both approaches have their advantages and disadvantages. Desmet et al. [10] call these *promptness strategy* and *loyalty strategy* respectively, giving examples of the use-

[21] These constructs need not be scattered throughout application code if they are encoded as CLOS *before*, *after*, or *around* methods.

fulness of both. The former refers to immediate reaction to context changes; the latter refers to delayed reaction to context changes (or no reaction at all), to avoid behavioural inconsistencies that could result from an immediate reaction.

The context-specific accessors of *AmOS* resemble the *layered accessors* of ContextL [7]. In both cases, observed object attributes can be different when consulted from different perspectives (contexts). However, once an attribute is read, the value that is obtained is oblivious to context —it continues to be the same when it is passed around in method invocations. This contrasts with *contextual values* [11], in which the value itself is sensitive to context. Contextual values add one more degree of sensitivity to context, which can be complementary to context-specific accessors.[22]

9 Conclusions and Future Work

Applications for Ambient Intelligence and Context-Oriented Programming require dynamic adaptation of behaviour according to the current physical and logical context in which the system is running. We have developed the Ambient Object System (*AmOS*), a simple yet flexible and expressive object model that aims at meeting the requirements of context adaptability. A few core concepts suffice to support fundamental abstractions such as activations, closures and methods, and more innovative abstractions such as contexts and behaviour dependency on contexts.

The system presented in this paper is fully operational and can be downloaded from `http://ambience.info.ucl.ac.be`. The same website offers related publications as well as some help on how to start programming in *AmOS*. Further information on *AmOS* is given in González [14].

In designing *AmOS* we have been mindful of future extensions to add concurrency and distribution. In particular, we are planning to extend *AmOS* with actor-based concurrency and dataflow synchronisation by means of asynchronous messages and futures.[23] To this end, we will probably borrow concepts from AmbientTalk [8] and Oz [30]. Regarding security, we need to assess the appropriateness of contexts (dynamic method scopes) as a simple visibility mechanism [27]. On the methodological side, we will be exploring the relationship of our approach to Feature-Oriented Domain Analysis [18] and the more recent and specific Context-Oriented Domain Analysis [9]. In Section 6.3 we touched upon feature interaction. We still have to develop a systematic way of dealing with feature interaction in *AmOS* (methodologically, or by means of dedicated language abstractions).

[22] A contextual value can be seen as a kind of *cell* object, whose main operations, "read value" and "write value", are performed implicitly for the sake of readability and ease of use. In a context-oriented language, these operations may depend on context.

[23] This requires first-class messages, which we have not incorporated in *AmOS* yet.

10 Acknowledgements

This work has been supported by the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), and by the Interuniversity Attraction Poles (IAP) Programme of the Belgian State, Belgian Science Policy. We thank the anonymous reviewers, including those of the 1st European Lisp Symposium, for their useful comments on earlier versions of this paper.

References

[1] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 69–82. ACM Press, 1996.

[2] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56. Springer-Verlag, 1992.

[3] Craig Chambers, David Ungar, and E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, 1989.

[4] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *Transactions on Programming Languages and Systems*, 28(3), May 2006.

[5] Pascal Costanza. Dynamically scoped functions as the essence of AOP. *ACM SIGPLAN Notices*, 38(8):29–36, 2003.

[6] Pascal Costanza, Charlotte Herzeel, Jorge Vallejos, and Theo D'Hondt. Filtered dispatch. In *Proceedings of the Dynamic Languages Symposium*. ACM Press, July 2008. Co-located with ECOOP'08.

[7] 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. Co-located with OOPSLA'05.

[8] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming. In *Companion to the annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 31–40. ACM Press, 2005.

[9] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-oriented domain analysis. In Boicho Kokinov, Daniel C. Richardson, Thomas R. Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context*, Lecture Notes in Computer Science, pages 178–191. Springer-Verlag, 2007.

[10] Brecht Desmet, Jorge Vallejos, Pascal Costanza, and Robert Hirschfeld. Layered design approach for context-aware systems. In *Proceedings of 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2007)*, pages 157–165. Technical Report at Irish Software Engineering Research Centre (Lero), January 2007.

[11] Éric Tanter. Contextual values. In *Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, 2008.

[12] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211. Springer-Verlag, 1998.

[13] Brian Foote, Ralph E. Johnson, and James Noble. Efficient multimethods in a single dispatch language. In *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 3586, pages 337–361. Springer-Verlag, 2005.

[14] Sebastián González. *Programming in Ambience: Gearing Up for Dynamic Adaptation to Context*. PhD thesis, Université catholique de Louvain, October 2008.

[15] 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. Colocated with OOPSLA'07.

[16] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *ACM SIGPLAN Notices*, 28(10):411–428, 1993.

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

[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, November 1990.

[19] Gregor Kiczales. Towards a new model of abstraction in the engineering of software. In *Proceedings of IMSA Workshop on Reflection and Meta-Level Architectures*, 1992.

[20] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Norman Meyrowitz, editor, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 214–223. ACM Press, 1986.

[21] Mark Miller and Jonathan S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In Vijay A. Saraswat, editor, *Advances in Computing Science, Proceedings of the Asian Conference on Programming Languages and Distributed Computation*, volume 2896 of *Lecture Notes in Computer Science*, pages 224–242. Springer-Verlag, 2003.

[22] James Noble, Antero Taivalsaari, and Ivan Moore, editors. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag, 1999.

[23] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 419–443. Springer-Verlag, 1997.

[24] Lee Salzman and Jonathan Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In Andrew P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming*, LNCS 3586, pages 312–336. Springer-Verlag, 2005.

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

[26] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for Self. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 303–330. Springer-Verlag, 1995.

[27] Randall B. Smith and David Ungar. A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems*, 2(3):161–178, 1996.

[28] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 38–45. ACM Press, 1986.

[29] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242. ACM Press, 1987.

[30] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[31] Pamela Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–29, 1993.