Context-Oriented Algorithmic Design

Bruno Ferreira
Instituto Superior Técnico/INESC-ID
Lisbon, Portugal
bruno.b.ferreira@tecnico.ulisboa.pt

António Menezes Leitão
Instituto Superior Técnico/INESC-ID
Lisbon, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT
Currently, algorithmic approaches are being introduced in several areas of expertise, namely Architecture. Algorithmic Design (AD) is an approach for architecture that allows architects to take advantage of algorithms to produce complex forms for their projects, to simplify the exploration of variations, or to mechanize tasks, including those related to analysis and optimization of designs. However, architects might need different models of the same project for different kinds of analysis, which tempts them to extend the same code base for different purposes, typically making the code brittle and hard to understand. In this paper, we propose to extend AD with Context-Oriented Programming (COP), a programming paradigm based on context that dynamically changes the behavior of the code. To this end, we propose a COP library and we explore its combination with an AD tool. Finally, we implement a case study with our approach, and discuss the advantages and disadvantages.

CCS CONCEPTS
• Software and its engineering → Object oriented languages;

KEYWORDS
Context-Oriented Programming, Algorithmic Design, Racket, Generative Design

1 INTRODUCTION
Nowadays, Computer Science is being introduced in several areas of expertise, leading to new approaches in areas such as Architecture. Algorithmic Design (AD) is one of such approaches, and can be defined as the production of Computer-Aided Design (CAD) and Building Information Modeling (BIM) models through algorithms [9, 25]. This approach can be used to produce complex models of buildings that could not be created with traditional means, and its parametric nature allows an easier exploration of variations.

Due to these advantages, AD started to be introduced in CAD and BIM applications, which led to the development of tools that support AD programs, such as Grasshopper [26]. However, with the complexity of the models came the necessity of analyzing the produced solutions with analysis tools. For this task, the geometrical models are no longer sufficient, as analysis software usually requires special analytical models, that are different from geometrical models and can hardly be obtained with import/export mechanisms due to errors. These requirements lead to the production of several models, which have to be kept and developed in parallel, involving different development lines that are hard to manage and to keep synchronized. This complex workflow proves that current solutions are not sufficient [36].

Some tools like Rosetta [24] are already trying to address these issues by offering portable AD programs between different CAD applications and, more recently, BIM applications [7, 8] and analysis tools [22]. Nevertheless, this tool does not offer a unifying description capable of producing both the geometrical and the analytical models with the same AD program, which can lead to the cluttering of the current program in an effort to reduce the number of files to maintain.

To solve these problems, we propose the use of COP to develop a computational model capable of adapting itself to the required context, which in this case is defined by the requirements of modeling applications and analysis tools, allowing the production of different models with a change of context.

1.1 Context-Oriented Programming
COP was first introduced as a new programming approach that takes the context into account [10]. According to a more recent depiction of this approach, COP aims to give users ways to deal with context in their programs, making it accessible to manipulation with features that are usually unavailable in mainstream programming languages [16].

With this approach, users can express different behaviors in terms of the context of the system. The context is composed of the actors of the system, which can determine how the system is used, the environment of the system, which can restrict or influence its functionality, and the system itself, whose changes might lead to different responses.

Although there are different implementations of COP, which will be presented later in this paper, according to [16] necessary properties must be addressed by all of them. These are:

• behavioral variation: implementations of behavior for each context;
• layers: a way to group related context-dependent variations;
• activation and deactivation of layers: a way to dynamically enable or disable layers, based on the current context;
• context: information that is accessible to the program and can be used to determine behavioral variations;
• scoping: the scope in which layers are active or inactive and that can be controlled.

With these features, layers can be activated or deactivated dynamically in arbitrary places of the code, resulting in behaviors that fit the different contexts the program goes through during its execution. If analyzed in terms of multi-dimensional dispatch
[34], it is possible to say that COP has four-dimensional dispatch, since it considers the message, the receiver, the sender, and the context to determine which methods or partial method definitions are included or excluded from message dispatch.

These method definitions are used to implement behavioral variations in layers, which can be expressed differently in the several implementations of COP. In some, the adopted approach is known as class-in-layer, in which layers are defined outside the lexical scope of modules [2], in a manner similar to aspects from Aspect-Oriented Programming (AOP) [20]. In others, a layer-in-class approach is used, having the layer declarations within the lexical scope of the modules.

Although these concepts are used to define COP, each implementation of the paradigm might include additional features and concepts, depending on the programming language. Some of these differences will be discussed later in the paper.

1.2 Objectives
The main objectives of this paper are: (1) present and compare the different implementations for COP that have been proposed by the research community, and (2) present a simple case study that shows how COP can be applied to AD. The case study consists of a previously developed AD program that we re-implemented with our proposed solution and then used to produce different models according to different contexts. Finally, the results of our solution are compared to the ones obtained in the previous version of the code.

2 RELATED WORK
In this section, we introduce several paradigms that served as basis for COP, namely AOP, Feature-Oriented Programming (FOP), and Subject-Oriented Programming (SOP), as well as an overview of the several implementations of COP that have been proposed throughout the years.

2.1 Aspect-Oriented Programming
Most programming paradigms, such as Object-Oriented Programming (OOP), offer some way to modularize the concerns necessary to implement a complex system. However, it is common to encounter some concerns that do not fit the overall decomposition of the system, being scattered across several modules. These concerns are known as crosscutting concerns.

AOP was created to deal with these crosscutting concerns, introducing ways to specify them in a modularized manner, called aspects. Aspects can be implemented with proper isolation and composition, making the code easier to maintain and reuse [20]. Using AOP, it is possible to coordinate the crosscutting concerns with normal concerns, in well-defined points of the program, known as join points.

In addition to these concepts, AOP implementations, such as AspectJ, introduce more concepts, such as pointcuts, which are collections of join points and values at those join points, and advice, which are method-like implementations of behavior attached to pointcuts [21]. An aspect is then a module that implements a crosscutting concern, comprised of pointcuts and advice.

AspectJ also offers cfow constructs, that allow the expression of control-flow-dependent behavior variations, making it possible to conditionally compose behavior with if pointcuts [21]. If we compare this with the basic description of COP, we can see several similarities, making it possible to define behavior that depends on a given condition, which can, in itself, be considered a context. However, while AOP aims to modularize crosscutting concerns, this is not mandatory in COP, since the use of layer-in-class approaches scatters the code across several modules.

In addition, COP allows the activation and deactivation of layers in arbitrary pieces of code, while AOP triggers pointcuts at very specific join points that occur in the rest of the program [16]. This makes COP more flexible in dealing with behavioral variation.

2.2 Feature-Oriented Programming
Feature-Oriented Programming (FOP) is an approach that was originally proposed to generalize the inheritance mechanisms offered by OOP. With FOP, users create their objects by composing features, instead of defining traditional classes. This process is called class refinement [27].

The features introduced in FOP work in a very similar way to mixins, a mechanism that allows the specification of methods to be used by other classes [35]. Mixin layers can also be defined, consisting of a mixin that encapsulates another mixin, and whose parameters encapsulate all the parameters of the inner mixin.

With mixins, users can increment existing classes, which is similar to what features allow. However, with features, overriding is not used to define the functionality of a subclass, but it is used as a mechanism to resolve dependencies between features [27]. Lifters are also used for this purpose. They are functions that lift one feature to the context of another. Although this feature is similar to method overriding used in inheritance, lifters depend on two features and are implemented as a separate entity.

By combining features, it is possible to create objects with a very specific functionality based in already existing features, something that would have to be done with inheritance in traditional OOP. This combination provides higher modularity, flexibility, and reusability, since each feature is an entity in itself.

FOP also shares the idea of supporting behavioral variations of the original program with the composition mechanism, which is similar to how layers in COP work. However, while COP offers the possibility of using these variations dynamically, through the activation and deactivation of layers, FOP focuses on selecting and combining behavior at compile-time [29].

2.3 Subject-Oriented Programming
SOP is a programming paradigm that introduces the concept of subject to facilitate the development of cooperative applications. In this paradigm, applications are defined by the combination of subjects, which describe the state and behaviors of objects that are relevant to that subject [13].

The Subjects’ goal is to introduce a perception of an object, such as it is seen by a given application. Subjects do so by adding classes, state, and behavior, according to the needs of that application. By doing this, each application can use a shared object through the
operations defined for its subject, not needing to know the details of the object described by other subjects [13].

Subjects can also be combined in groups called compositions, which define a composition rule, explaining how classes and methods from different subjects can be combined. These subjects and compositions can then be used through subject-activation, which provides an executable instance of the subject, including all the data that can be manipulated by it.

Due to the introduction of all these concepts, SOP offers what is known as Subjective Dispatch [34]. Subjective Dispatch extends the dispatch introduced by OOP, by adding the sender dimension, in addition to the message and the receiver. This type of dispatch was later expanded by COP, which introduces a dimension for the context, as mentioned previously.

It is possible to see that, similarly to the other analyzed paradigms, SOP also supports behavior variations in the form of subjects. However, if we consider that each subject might have different contexts of execution, we need an extra dimension for dispatch, which is what COP offers.

### 2.4 Context-Oriented Programming Implementations

COP was proposed as an approach that allows the user to explore behavioral variations based on context. Concepts such as layers and contexts are present in all implementations of this approach. Nevertheless, each one can address the concepts differently, sometimes due to the support of the host language in which the COP constructs are implemented. In this section we present the different implementations available.

#### 2.4.1 ContextL

ContextL was one of the first programming language extensions to introduce support for COP. It implements the features discussed previously by taking advantage of the Common Lisp Object System (CLOS) [5].

The first feature to be considered is the implementation of layers, which are essential to implement the remaining features available in ContextL [6]. These layers can be activated dynamically throughout the code, since ContextL uses an approach called Dynamically Scoped Activation (DSA), where layers are explicitly activated and a part of the program is executed under that activation. The layer is active while the contained code is executing, becoming inactive when the control flow returns from the layer activation.

Regarding the activation of multiple layers, it is important to note that the approach introduced in ContextL, as well as in other implementations that support DSA, follows a stack-like discipline. Also, in ContextL, this activation only affects the current thread.

By taking advantage of layers, it is then possible to define classes in specific layers, so that the classes can have several degrees of detail in different layers, introducing behavior that will only be executed when specific layers are activated. The class behavior can also be defined with layered generic functions. These functions take advantage of the generic functions from CLOS, and are instances of a generic function class named layered-function [6].

In addition, ContextL supports contextual variations in the definition of class slots as well. Slots can be declared as layered, which makes the slot accessible through layered-functions. This feature introduces slots that are only relevant in specific contexts.

By looking at the constructs implemented in ContextL, it is possible to conclude that behavioral variations can be implemented in specific classes or outside of them. This means that ContextL supports both layer-in-class and class-in-layer approaches. The former allows the definition of partial methods to access private elements of enclosing classes, something that the latter does not support, since class-in-layer specifications cannot break encapsulation [2].

Finally, it should be noted that ContextL follows a library implementation strategy: it does not implement a source-to-source compiler, and all the constructs that support the COP features are integrated in CLOS by using the Metaobject Protocol [19].

#### 2.4.2 PyContext and ContextPy

PyContext was the first implementation of COP for the Python programming language. Although it includes most of the COP constructs in a similar manner to the other implementations, PyContext introduces new mechanisms for layer activation, as well as to deal with variables.

Explicit layer activation is an appropriate mechanism for several problems but, sometimes, this activation might violate modularity. Since the behavioral variation may occur due to a state change that can happen at any time during the program execution, the user needs to insert verifications in several parts of the program, increasing the amount of scattered code. To deal with this problem, PyContext introduces implicit layer activation. Each layer has a method active, which determines if a layer is active or not. This method, in combination with a function layers.register_implicit, allows the framework to determine which layers are active during a method call, in order to produce the correct method combination [37].

Regarding variables, PyContext offers contextual variables, which can be used with a with statement in order to maintain their value in the dynamic scope of the construct. These variables are called dynamic variables. These variables are globally accessible, and their value is dynamically determined when entering the scope of a with construct [37]. In conjunction with specific getters and setters, it is possible to get the value of the variable, change it in a specific context, and then have it restored when exiting the scope of that context. It is important to note that this feature is thread-local.

As for the other features, PyContext does not modify the Python Virtual Machine, being implemented as a library. Layers are implemented using meta-programming, and layer activation mechanisms take advantage of Python’s context handlers. As for the partial definition of methods and classes, PyContext follows a class-in-layer approach.

More recently, ContextPy was developed as another implementation of COP for the Python language. This implementation follows a more traditional approach to the COP features, offering DSA, using the with statement, which follows a stack-like approach for method composition. For partial definitions, ContextPy follows a layer-in-class approach, taking advantage of decorators to annotate base methods, as well as the definitions that replace those methods when a specific layer is active [17]. Finally, similarly to PyContext, ContextPy is offered as a library that can be easily included in a Python project.

#### 2.4.3 ContextJ

ContextJ is an implementation of COP for the Java programming language, and one of the first implementations of this approach for statically typed programming languages. Before
this implementation, there were two proof of concepts implemented in Java, namely ContextJ* [16] and ContextLogicAJ [1]. The first is a library-based implementation that does not offer all COP functionalities, while the second is an aspect-oriented pre-compiler that improves the features given by ContextJ* and offers new mechanisms, such as indefinite layer activation. Indefinite activation requires the user to explicitly activate and deactivate layers, in order to obtain the desired layer composition.

ContextJ is a source-to-source compiler solution that introduces all the concepts of COP in Java by extending the language with the layer, with, without, proceed, before and after terminal symbols [4]. Layers are included in the language as a non-instantiable type, and their definitions follows a layer-in-class approach. Each layer is composed of an identifier and a list of partial method definitions, whose signature must correspond to one of the methods of the class that defines the layer. Also, to use the defined layers, users must include a layer import declaration on their program, in order to make the layer type visible.

As for partial method definitions, they override the default method definition and can be combined, depending on the active layers. The before and after modifiers can also be used in partial method definitions, in order to include behavior that must be executed before and after the method execution. In addition, the proceed method can be used to execute the next partial definition that corresponds to the next active layer, allowing the combination of behavioral variations [4].

Regarding layer activation, ContextJ supports DSA by using a with block. Layers are only active during the scope of the block, and the activation is thread-local. With blocks can be nested, and the active layer list is traversed according to a stack approach. This approach, in combination with the proceed function, allows the user to compose complex behavior variations. In addition, it is possible to use the without block to deactivate a layer during its scope, in order to obtain a composition without the partial method definitions of that specific layer.

Finally, ContextJ also offers a reflection Application Programming Interface (API) for COP constructs. It includes classes for Layer, Composition, and PartialMethod, along with methods that support runtime inspection and manipulation of these concepts.

2.4.4 Other COP Implementations. The implementations described in the previous sections present some of the major strategies and features that are currently used with COP. Nevertheless, there are more implementations for other languages, which we briefly describe in this section.

Besides ContextJ, ContextJ*, and ContextLogicAJ, there are other implementations of COP for Java, namely: JCop [3] and EventCJ [18], which use join-point events to switch layers; cj [31], a subset of ContextJ that runs on an ad hoc Java virtual machine; and JavaCtx [28], a library that introduces COP semantics by weaving aspects.

There are also COP implementations for languages such as Ruby, Lua, and Smalltalk, namely ContextRx [32], ContextLua [38], and ContextTs [15] respectively. ContextRx introduces reflection mechanisms to query layers, while ContextLua was conceived to introduce COP in games. ContextTs follows the more traditional COP implementations, such as ContextL. ContextScheme1, for the Scheme programming language, also follows an implementation similar to ContextL.

In addition, some implementations, such as ContextErlang, introduce COP in different paradigms, like the actor model [14]. ContextErlang also introduces different ways to combine layers, namely per-agent variation activation and composition [30].

Regarding layer combination and activation, there are also implementations that offer new strategies that differ from dynamic activation. One example is ContextJS [23] that offers a solution based on open implementation, in which layer composition strategies are encapsulated in objects. These strategies can add new scoping mechanisms, disable layers, or introduce a new layer composition behavior that works better with a domain-specific problem [23].

More recently, Ambience [12], Subjective-C [11], and Lambic [35] were developed. Ambience uses the amOS language and context objects to implement behavioral variations, with the context dispatch made through multi-methods. Subjective-C introduces a Domain Specific Language (DSL) that supports the definition of constraints and the activation of behaviors for each context. Finally, Lambic is a COP implementation for Common Lisp that uses predicate dispatching to produce different behavioral variations.

In the next section we present a comparison between all these implementations, as well as the advantages and disadvantages of using each one.

2.5 Comparison

Table 1 shows a comparison between the analyzed COP implementations.

As it is possible to see, most of the analyzed implementations are libraries, with source-to-source compilers being mostly used in statically typed programming languages. The library implementation has advantages when trying to add COP in an already existing project, since it does not change the language and uses the available constructs. On the other hand, source-to-source compilers, such as ContextJ, introduce new syntax that simplifies the COP mechanics, as well as possible advantages regarding performance.

As for layer activation, the most common strategy is DSA. However, to increase flexibility, some solutions introduce indefinite activation, global activation, per agent activation or, in the case of ContextJS, an open implementation, allowing users to implement an activation mechanism that best fits the problem they are solving. Although DSA is appropriate for most problems, other strategies might be best suited for multi-threaded applications or problems whose contexts depend on conditions that cannot be captured with the default layer activation approach.

Finally, regarding modularization, it is possible to see that most implementations use the class-in-layer or the layer-in-class approach. The first one allows users to create modules with all the concerns regarding a specific context, while the latter places all the behaviors on the class affected by the contexts. Hence, class-in-layer reduces code scattering, while layer-in-class can simplify program comprehension. There are implementations that support both approaches, such as ContextL, but usually the supported approach is restricted by the features of the language. Nevertheless, there are cases, such as ContextPy and PyContext, that take advantage

1http://p-cos.net/context-scheme.html
of the same programming language but follow different principles regarding the COP concepts.

All these implementations support the COP paradigm, although they offer different variations of the relevant concepts. Choosing the most appropriate implementation requires a careful examination of their distinct features, and how they help in fulfilling the requirements of the problem at hand.

3 CONTEXT-ORIENTED ALGORITHMIC DESIGN

In this section, we propose to combine COP with AD, introducing what we call Context-Oriented Algorithmic Design. Since it is common for architects to produce several different models for the same project, depending on the intended use (e.g., for analysis or rendering), we define these different purposes as contexts. By doing this, it is possible to explicitly say which type of model is going to be produced.

In addition, we introduce definitions for the design primitives using COP as well. For each primitive, we can define different behavioral variations, depending on the model we want to produce. For example, some analysis models require surfaces instead of solids, a primitive definition of a Wall would produce a box in a 3D context and a simple surface in an analysis context.

Finally, since COP allows the combination of layers, we can take advantage of that to combine additional concepts, such as Level of Detail (LOD) with the remaining ones. This combination allows more flexibility while exploring variations, since it not only supports the exploration in several contexts, but also the variation of LOD inside the same context. This might be useful, e.g., for architects that want to have less detail in certain phases to obtain quicker results.

### Table 1: Comparison between the COP implementations. DSA stands for Dynamically Scoped Activation, LIC for layer-in-class, and CIL for class-in-layer. Lambic uses predicate dispatching instead of layers, so the last two columns do not apply. Adapted from [29]

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Base Language</th>
<th>Layer Activation</th>
<th>Modulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContextL</td>
<td>Common Lisp</td>
<td>Library</td>
<td>DSA</td>
</tr>
<tr>
<td>ContextScheme</td>
<td>Scheme</td>
<td>Library</td>
<td>DSA</td>
</tr>
<tr>
<td>ContextErlang</td>
<td>Erlang</td>
<td>Library</td>
<td>Per-agent</td>
</tr>
<tr>
<td>ContextJS</td>
<td>JavaScript</td>
<td>Library</td>
<td>Open Implementation</td>
</tr>
<tr>
<td>PyContext</td>
<td>Python</td>
<td>Library</td>
<td>DSA, Implicit Layer</td>
</tr>
<tr>
<td>ContextPy</td>
<td>Python</td>
<td>Library</td>
<td>Activation</td>
</tr>
<tr>
<td>ContextJ</td>
<td>Java</td>
<td>Source-to-Source Compiler</td>
<td>DSA</td>
</tr>
<tr>
<td>JCop</td>
<td>Java</td>
<td>Source-to-Source and Aspect Compiler</td>
<td>DSA</td>
</tr>
<tr>
<td>EventCJ</td>
<td>Java</td>
<td>Source-to-Source and Aspect Compiler</td>
<td>DSA</td>
</tr>
<tr>
<td>Javactx</td>
<td>Java</td>
<td>Library and Aspect Compiler</td>
<td>DSA</td>
</tr>
<tr>
<td>ContextR</td>
<td>Ruby</td>
<td>Library</td>
<td>DSA</td>
</tr>
<tr>
<td>ContextLua</td>
<td>Lua</td>
<td>Library</td>
<td>DSA</td>
</tr>
<tr>
<td>ContextS</td>
<td>Smalltalk</td>
<td>Library</td>
<td>DSA, indefinite activation</td>
</tr>
<tr>
<td>Ambience</td>
<td>AmOS</td>
<td>Library</td>
<td>DSA, global activation</td>
</tr>
<tr>
<td>Lambic</td>
<td>Common Lisp</td>
<td>Library</td>
<td>-</td>
</tr>
<tr>
<td>Subjective-C</td>
<td>Objective-C</td>
<td>Preprocessor</td>
<td>Global Activation</td>
</tr>
</tbody>
</table>

3.1 Implementation

To test our solution we created a working prototype, taking advantage of Khepri, an existing implementation of AD, and ContextScheme to introduce the COP concepts. Khepri is a portable AD tool, similar to Rosetta [24], that allows the generation of models in different modeling back-ends, such as AutoCAD or Revit, and offers a wide range of modeling primitives for the supported applications. This tool offers a native implementation in Racket, a pedagogical programming language with support for COP, making it easier to extend.

As for the choice of the COP implementation to use, we decided for an implementation of ContextScheme that we adapted for Racket. This implementation is a library, making it easier to include in existing projects and tools, such as Khepri. Also, since AD programs are usually single-threaded, and we can easily indicate the scope to be affected by the context, DSA is a good approach to solve the problem. Both features are supported by ContextScheme.

Having these two components, we implemented a new library, that extends Khepri, and introduces design elements with contextual awareness. Since the behavioral variations have to produce results on the selected modeling tool, this new layer uses Khepri’s primitive functions to do so. The functions to use depend on the context. For instance, the program uses those that produce surfaces in analysis contexts, and those that produce solids on 3D contexts. For each new modeling element of our library, we define functions with basic behavior and a variation for each possible context. The available contexts are implemented as layers in the library as well.

Listing 1 shows a simplified definition of the wall function. This definition has a default behavior, and variations for a 3D, 2D, and analysis contexts, which are identified by the layers used as
parameters. In each of these functions, Khepri modeling functions are used, in order to produce the results in the modeling tools.

Listing 1: Definition of 3D and 2D walls.

```scheme
(define-layered wall)
(lambda (...)
  (box ...)))

(define-layered (wall 2D)
(lambda (...)
  (rectangle ...)))

(define-layered (wall analysis)
(lambda (...)
  (surface ...)))
```

In the next section, we introduce a case study, that was produced with this new library, in order to evaluate our approach.

4 CASE STUDY

For the evaluation of our COP library we used a model of a shopping mall, originally used for evacuation simulations. The model was produced with an algorithmic solution, which we modified to include our library. This is a simple case study that uses few geometrical elements, namely doors, walls, and a floor, which the new library already supports.

We chose this case study because the original implementation required a plan view of the model, which had to be produced in addition to the usual 3D view. To take advantage of the same algorithm, the original developers included two implementations of the shop function, which produce each of the shops that compose the mall, and are used by the rest of the algorithm. One implementation is a 2D version that created lines, and the other is a 3D version that created solids.

In order to switch between them, the authors commented a couple lines of code and changed some variables as well. This approach has several disadvantages, namely the need to modify lines of code when it is necessary to change the type of model, and having to comment and uncomment several lines of code when we want to change from 2D to 3D. Both of these tasks are error prone, since developers might forget to do some of them.

Listing 2 shows a simplified definition of both 2D and 3D versions of the shop function, as well as the commented line of code correspondent to the 2D version, having the 3D version activated in the next line. The 2D version uses functions that generate 2D shapes, such as rectangles and lines, and the 3D version uses functions that generate solids.

Listing 2: Original version.

```scheme
(define (shop-2d ...)
  (...)
  (rectangle ...)
  ...)
)

(define shop shop-2d)

(define shop shop-3d)
```

Our COP-based solution eliminates the aforementioned problems. By adding our library, we re-implemented parts of the algorithm, namely the shop function. Since we have different implementations for the elements, such as walls, available in different contexts, we do not require two versions of the same function, and can implement just one. Listing 3 shows a simplified definition of the COP version of the shop function, using the wall and door functions of our library.

Listing 3: COP implementation of the shop function.

```scheme
(define (shop ...
  (...)
  ((wall) ...
    ...) ...
  (door) ((wall) ...
    ...) ...
  )
)
```

To switch between contexts, we eliminated the commented lines of code and introduced a `with-layers` construct, which receives the layer corresponding to the model we want to produce, and the expression that generates the entire shopping mall. For example, the `with-layers` can receive a layer corresponding to the 3D view and have a call to the `mall` function in its scope.

Since we wanted to produce a plan view and a 3D model, and those correspond to layers that we support in the library, we could generate both of them by introducing 3D or 2D as arguments of the `with-layers` construct. The results can be seen in figures 1 and 2.

Figure 1: 3D model of the shopping mall, produced with COP in AutoCAD.

In addition, since we support a layer that produces only surfaces for analysis purposes, namely radiation analysis, we were able to produce another model simply by changing the context. By using analysis as argument for the `with-layers` construct, we...
produced a model for analysis (visible in Figure 3) without any changes to the algorithm.

With our solution, we were able to reduce the code that produces the models and introduce a more flexible way to both change the context and produce different views of the model. This did not require any additional functions, except the with-layers construct. In addition, by simply expanding the library and introducing new contexts, our algorithms are capable of producing new models without any changes, which would require additional code in the traditional approach.

Finally, there is another advantage in using our proposed solution, in comparison to the approach used by the original developers. In the original solution, one of the functions was chosen before the execution of the program, using the same function for all the generated elements, hence the same context. Changing the context would require the developer to stop the execution and change the code. On the other hand, with the COP version, layers are activated and deactivated dynamically, meaning that different parts of the program can be executed in different contexts. This feature offers more flexibility to developers, allowing the production of more complex models, where some elements can be represented in a simplified form, and others can be represented with more detail.

This is useful in phases that only require further development of a group of elements, not needing detail in the remaining elements.

4.1 Evaluation

As it was possible to see in the previous section, ContextScheme and Racket allowed us to write context-dependent code that simplified the application, allowing the production of several models for different contexts. However, the resulting code can still be improved, making it easier to write, understand, and maintain.

For example, by examining listing 3, we can see an invocation of the wall and door functions before passing the actual arguments. This happens because ContextScheme uses higher-order functions that return the appropriate function for each context. However, if we chose ContextL instead, this would not be necessary, as the contextual functions can be used directly as normal functions, which simplifies the code.

Another relevant dimension is the way we interact with the contexts. In Ambience, context objects are implicit parameters and global activation is an option. This type of activation would allow us to activate the context and then write all the code we want to execute, instead of including it inside the scope of a with-layers construct. Nevertheless, this would require the user to deactivate and activate contexts explicitly in the code, so it is not clear if this option would simplify the experience of the user.

Finally, regarding performance, we have not yet done a comparison between the available COP implementations. However, due to the overheads involved in the creation of the geometric models, the impact of the COP implementation is negligible. For this reason, when used just for the implementation of the modeling operations, the COP implementation performance can be ignored.

5 CONCLUSIONS

Currently, algorithmic approaches are used in Architecture to create complex models of buildings that would otherwise be impossible to produce. Moreover, AD also simplifies and automates several tasks that were error-prone and time-consuming, and allows an easier exploration of variations. Nevertheless, when architects want to use analysis tools, or simply produce different views of the same model, they need additional algorithms, increasing the versions of code to maintain.

In this paper, we explore the combination of AD with COP, a paradigm that dynamically changes the behavior of the code depending on the active context, which is implemented with layers. There are several lines of research related to COP, which led to multiple implementations for different programming languages, namely ContextL, ContextJ, and ContextPy, among others, all of which have different features and advantages.

In our solution, we took advantage of a COP library that uses DSA, and a class-in-layer approach. All these features fit the needs of AD problems, and the use of Racket simplifies the introduction of COP in existing tools, such as Khepri.

To test our solution, we used our COP library in an existing AD program that produced the model of a shopping mall for simulation purposes. The program included multiple definitions of the same function to produce different views of the model, which were activated by commenting and uncommenting code. The program
was re-implemented with COP, which eliminated the multiple definitions and the commented code. The use of our approach allowed the production of the model for several different contexts without additional changes in the program.

With this case study, we can conclude that COP can be combined with AD and it can be useful when exploring different views of the models, which require different behaviors from the same program.

6 FUTURE WORK
As future work, we will continue to expand our library with more building elements. We will also introduce more contexts, giving users more layers for the production of different kinds of models.

In addition, we will explore the combination of layers in order to obtain more sophisticated results. One idea is to explore a LOD layer in combination with the other layers, in order to produce simpler models in an exploration phase, and more complex ones in later stages of development.

ACKNOWLEDGMENTS
This work was partially supported by national funds through Função para a Ciência e a Tecnologia (FCT) with reference UID/ CEC/50021/2013.

REFERENCES