Efficient Layered Method Execution in ContextAmber

Matthias Springer
Hasso Plattner Institute
University of Potsdam
matthias.springer@hpi.de

Jens Lincke
Hasso Plattner Institute
University of Potsdam
jens.lincke@hpi.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
hirschfeld@hpi.de

ABSTRACT

We present ContextAmber, a framework for context-oriented programming, in Amber Smalltalk, an implementation of the Smalltalk programming language that compiles to JavaScript. ContextAmber is implemented using metaprogramming facilities and supports global, object-wise, and scoped layer activation. Current COP implementations come at the expense of significantly reduced execution performance due to multiple partial method invocations and layer composition computations every time a layered method is invoked. ContextAmber can reduce this overhead by inlining partial methods, caching layer compositions, and caching inlined layered methods, resulting in a runtime overhead of about 5% in our vector graphics rendering benchmarks.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming;
D.3.4 [**Programming Languages**]: Processors—*code generation, optimization*

General Terms

Languages

Keywords

Context-oriented programming, partial method inlining, inlined layered method invalidation

1. INTRODUCTION

Layer-based context-oriented programming [6] is a way to modularize crosscutting concerns that can dynamically adapt their runtime behavior: layers can be activated and deactivated at runtime, making it hard to predict the sequence and nesting of invoked partial methods for a given layered method at compile time. Performance studies have shown that in static aspect-oriented programming, where aspect weaving is done at compile time, the runtime overhead can be as small as 1% [3], whereas in context-oriented programming, the performance decrease of layer-aware method dispatch is typically more than 75% [1].

Upon invocation of a layered method, a COP implementation without any further optimizations has to perform the following steps.

1. Compute the receiver's layer composition (*active layers*).
2. Find and invoke the partial method corresponding to the topmost layer having a partial method for the invoked base method.
3. Find and invoke the next partial method when a `proceed` call is encountered.

In this paper, we present optimizations that speed up the execution of COP applications: our implementation, ContextAmber, avoids unnecessary computations of layer compositions and reduces the number of method invocations by inlining partial methods. It is no longer necessary to find the next corresponding partial method since only a single fully inlined method is invoked. Note, that finding the next partial method can be expensive in other implementations, because it requires going through the collection of active layers and checking whether a layer has a method for the receiver's class or one of the receiver's super classes. Our main contribution is an evaluation of techniques for invalidating inlined methods.

ContextAmber is implemented using JavaScript/Smalltalk metaprogramming facilities and does not require changes to the underlying JavaScript interpreter or virtual machine. Most findings of this paper can be applied to all programming languages that support on-the-fly code generation like Ruby, Smalltalk, or JavaScript.

In the remainder of this paper, we discuss vector graphics rendering debugging which serves as a running example in this paper (Section 2). In Sections 3-5, we present the key ideas of our implementation, along with performance measurements in Section 6.

2. VECTOR GRAPHICS RENDERING DEBUGGING WITH COP

In this section, we will show how graphical user interfaces can be rendered in Amber Smalltalk using Athens, serving as a running example for the remainder of this work.

Athens[1] is a vector graphics library available for Pharo and Amber Smalltalk. It provides a simple API for drawing

[1] `http://smalltalkhub.com/#!/~Pharo/Athens`

COP'15 July 05 2015, Prague, Czech Republic.

ACM 978-1-4503-3654-3/15/07 . . . $15.00. http://dx.doi.org/10.1145/2786545.2786548.

geometric shapes such as rectangles and ellipses. It is also possible to define *paths* which consist of a number of path *segments*. A segment can be a line or any kind of mathematical curve, e.g., a Bézier curve. Once a path object is created, it can be shared in the application and be drawn an arbitrary number of times.

Consider, for example, that Athens is used to draw the user interface for an application. The user interface consists a small number of distinct user interface elements. All elements of a certain type are separate objects but share the same path objects for drawing, making it possible to enforce a consistent style and to change this style easily. For example, every time the application draws a button[2], it uses the path defined in Figure 1.

```
path := surface createPath: [ :builder |
    builder
        absolute;
        moveTo: 5@0;
        lineTo: 45@0;
        cwArcTo: 50@5 angle: 90;
        lineTo: 50@15;
        cwArcTo: 45@20 angle: 90;
        lineTo: 5@20;
        cwArcTo: 0@15 angle: 90;
        lineTo: 0@5;
        cwArcTo: 5@0 angle: 90 ].

canvas draw: path.
```

Figure 1: Drawing a rectangle with rounded corners in Athens.

Figure 2 shows how Athens draws paths. `AthensCanvas` is the entry point for all drawings and has methods for performing primitive operations such as moving the pen somewhere on the canvas. The `draw:` method can be used to draw complex shapes such as paths. It delegates the drawing to that path object which acts as a proxy here. A path contains a list of segments that is created when the path is created. During drawing, these segments are *replayed* using double dispatch. In our example, the path object simply forwards these commands to the canvas object[3].

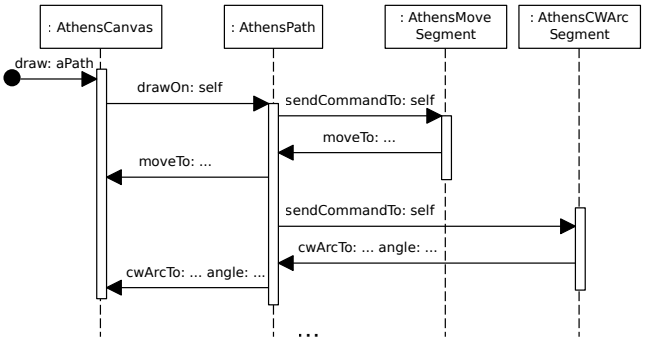


Figure 2: Drawing paths in Athens.

[2]To simplify this example, we assume that all buttons have the same size.

[3]It is possible to extend Athens for different drawing backends, requiring a different interface, in which case the path object is an adapter.

Control Point Layer.

To make it easier to find drawing bugs, a control point layer is defined, which draws control points for all segments of a path. Control points are, for example, start and end points of lines and curves, or Bézier control points.

```
moveTo: aPoint
    self proceed: aPoint.
    self drawControlPoint: endPoint.

drawControlPoint: aPoint
    canvas pushStyle.
    canvas fillStyle: 'rgba(0, 0, 0, 0.5)'.
    canvas fillRect: (aPoint - (5@5)
                        corner: 10@10).
    canvas popStyle.
```

Figure 3: Some partial methods for drawing control points.

Figure 3 shows some of the partial methods defined for the control point layer. `AthensPath` is the base class for these methods. ContextAmber allows programmers to add new methods to classes. For example, `drawControlPoint:` is not defined on `AthensPath`. By activating the control point layer on the path defined in Figure 1, it is possible to show control points only for buttons, but not for other user interface elements.

3. IMPLEMENTATION

ContextAmber is our framework for context-oriented programming and used to evaluate and benchmark the concepts presented in this paper. It runs on top of Amber Smalltalk[4], an implementation of the Smalltalk programming language that compiles to JavaScript.

Amber Smalltalk Object Model.

The Amber Smalltalk object model is minimalistic but follows closely the Smalltalk-80 object model. Every Smalltalk object is at the same time a JavaScript object. Instance-of and inheritance relationships are implemented using the JavaScript prototype hierarchy, i.e., every object has a prototype containing the instance methods for the corresponding class, and that object's prototype contains the instance methods of the super class. Message passing is implemented by JavaScript method calls, where colons in the selector are replaced with underscores.

When a Smalltalk method is compiled, its source code is first converted into an AST. Then a semantic analyzer performs checks and determines the type and scope of variables. The AST is then converted into an intermediate representation (IR) which is similar to the AST but distinguishes between local returns and non-local returns, for example. An IR tree can directly be transformed into JavaScript code.

ContextAmber performs method inlining on the AST level, making it possible to use the step-through debugger, which operates on the AST.

Layer Activation.

In ContextAmber, layers can be activated globally, within a certain scope, and for a single object. Since JavaScript is

[4]`http://amber-lang.net`

a single-threaded programming language, scoped layer activation can be treated like global layer activation[5], as long as the effect is reversed once the control flow leaves the scope. As will be described in Section 4, object-wise layer activation has far-reaching effects on method inlining, since different instances of the same class can have different layer compositions.

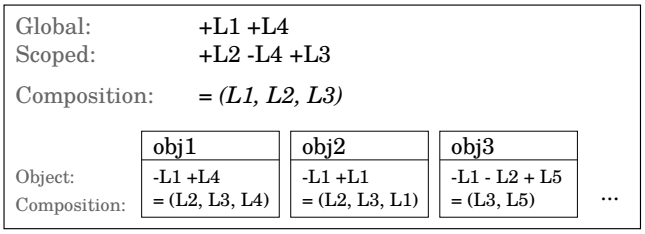


Figure 4: Layer Composition Calculation in ContextAmber.

Figure 4 shows how ContextAmber calculates an object's layer composition. Object-wise layer activation has a higher precedence than scoped layer activation, which has a higher precedence than global layer activation. In the rest of this work, we do not distinguish between global and scoped layer activation and refer to the global/scoped layer composition as the *Global Layer Composition* (italic text in Figure 4).

4. PARTIAL METHOD INLINING

In this section, we discuss ContextAmber's two variants for inlining partial methods and their implications.

4.1 Method Inlining

ContextAmber hooks into the Amber compilation process and inlines methods on the AST level. A visitor looks for `proceed` sends and replaces them with a `value` send to a block closure containing the sequence of instructions of the next partial method in the current layer composition. This approach takes care of name clashes of temporary variables, as opposed to just copying the instructions without a block closure. Return statements in inlined methods must be modified, such that they don't cause the entire method to end but just return from the block closure (local return).

Class-specific Method Inlining.

Inlined methods are stored on the prototype object for the class (like ordinary instance methods), i.e., the prototype that is shared by all instances of a class. This makes object-wise layer activation difficult, because ContextAmber might have to execute different inlined methods, depending on the object's layer composition.

Instance-specific Method Inlining.

Inlined methods are stored as attributes on the object itself, allowing objects with different layer compositions to have their own inlined methods. This is usually not supported by Smalltalk, but, in JavaScript, every object can have its own methods. Some programming languages have similar concepts: for example, Ruby supports object-specific methods using singleton classes (eigenclasses). In some other programming languages, instance-specific method inlining can be harder to implement using only metaprogramming facilities: an implementation would have to create a subclass containing the inlined method and change the object's class to that subclass. Alternatively, layered methods could be wrappers delegating calls to composition-specific item description objects [2], or look up methods in a separate dictionary.

Instance-specific inlining can be faster than class-specific inlining because no new methods have to be installed when a method is called on objects with different layer compositions alternately. However, having too many different layer compositions can affect caching and VM optimizations and slow down the system. Future versions of ContextAmber might automatically decide whether to use class-specific or instance-specific inlining and adapt that decision during runtime if necessary. Currently, the programmer has to decide.

[5]Amber Smalltalk does currently not support processes.

4.2 Method Cache

ContextAmber maintains a fixed-size method cache mapping layer compositions to inlined methods. Whenever an inlined method is requested, it is first looked up in the method cache. If an inlined method was not found in the cache, it is generated and added to the cache. Old inlined methods are evicted from the cache in a FIFO manner.

Method caches make instance-specific method inlining feasible even with a large number of objects, because the same inlined method can be shared by multiple objects.

5. INLINED METHOD INVALIDATION

Method inlining improves the execution performance of layered methods; however, the overhead of iterating through the layer composition and performing a number of partial method dispatches is just shifted to another point of time. Whenever we inline a method, we are speculating that the layer composition will remain constant for a while, but we have to validate that assumption at some point. Once it becomes apparent that an inlined method is no longer up to date, it is invalidated, which causes the inlined method to replace itself with an updated version when it is called again. In this section, we discuss how and when ContextAmber invalidates inlined methods (see Figure 5 for an overview).

5.1 Method Invalidation Reasons

The following list gives an overview of the events that can cause an inlined method to be invalidated.

- *Adding/removing a Partial Method.* If a partial method is added or removed for a layer that is currently active, ContextAmber might have to add or remove that partial method to corresponding inlined layered methods.
- *Calling a Method on a Different Object.* With object-wise layer activation, two different instances of the same class can have different layer compositions and, therefore, different inlined layered methods. A layered method that is inlined for the entire class has to invalidate itself when it is invoked on an object having a different layer composition than the one it was created for.
- *Layer Composition Change.* Inlined methods can be become outdated when a layer is activated or deactivated.

5.2 Invalidation on Invocation

Every inlined method contains an *update header* that determines if the method is up to date. If this check is positive, then the execution of the inlined method continues.

	instance-specific	class-specific	instance space overhead	update header runtime overhead[6]	composition change runtime overhead
Comparing signatures	✓	✓	no overhead	compute composition compare signature	no overhead
Caching signatures	✓	✓	boolean field integer field string field	compare dirty bit compare version (compare signature)	object: set dirty bit global: increment version
Detecting comp. changes without signatures	✓	✗	no overhead	compare signature	object: delete method global: compute global composition signature
Method generation on composition change	✓	✗	no overhead	no overhead	compute composition generate mult. methods

When using instance-specific method inlining, an additional method object is stored per instance[7].

Figure 5: Comparison of techniques for layered method invalidation and inlining.

Otherwise, a new version of the inlined method with the current layer composition is generated, installed, and executed. In this subsection, we give an overview of different update headers and when they are suitable.

Comparing Composition Signatures.

Every layer has a unique integer identifier. A layer composition can be uniquely represented by its layer composition signature (i.e., fingerprint [8]): a concatenation of the layers' IDs and separator characters. The most basic update header is shown in Figure 6.

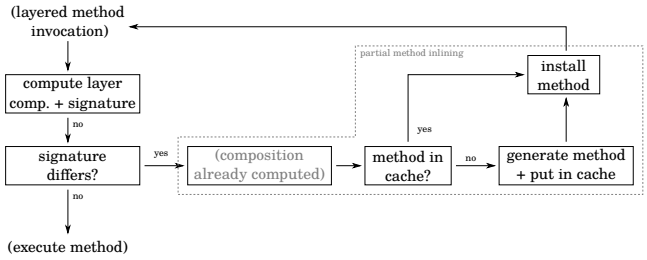


Figure 6: Update header for comparing layer composition signatures.

When an inlined layered method is invoked, the update header computes the receiver's layer composition and its layer composition signature and compares it with the layer composition signature at the point of time when the inlined method was generated. That value is hard-coded as a string literal in the update header's source code. If the signature values differ, then a new inlined method is created, installed, and called. A method cache is used to cache inlined methods and avoid unnecessary code generation.

Caching Composition Signatures.

The update header in Figure 6 has to compute the current layer composition every time the method is invoked. ContextAmber can speed up this step by caching layer composition signatures on a per-object basis. Recalculating the signature whenever a composition change is made is not suitable, because a global layer composition change can affect a large number of objects whose signatures would have to be updated (Figure 4). Instead, the composition signature becomes *outdated* whenever the layer composition is changed globally or on a per-object basis: in the former case, a version number is incremented on all base classes affected by the layer, indicating that a change was made that affects all of its instances. In the latter case, the object is marked as dirty, indicating that only the layer composition for that single object changed. The update header uses the cached composition signature if the receiver is not marked dirty and its version number equals its class' version number. Otherwise, the layer composition is recomputed, along with its signature (Figure 7).

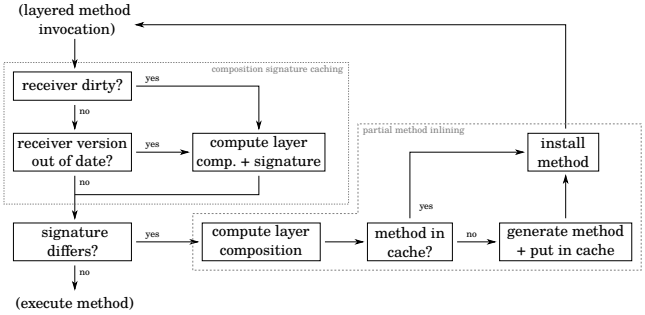


Figure 7: Update header for caching layer composition signatures.

In total, ContextAmber stores three additional fields on every object with a layered method[8]: the cached layer composition signature (string), the version number (integer) of the object's class at the point of time when the layer signature was cached, and a dirty bit (boolean). These three fields are refreshed whenever the composition signature is recomputed.

Note, that composition signature is both hard-coded in the method source code and stored as a field on every object having layered methods. These two values are being compared in the update header. ContextAmber uses this technique for class-specific method inlining.

[6]We consider the average case where the layer composition was not changed (or invariant layer composition change sequences).

[7]Method caches make it easy to share the same method object among multiple instances.

[8]These fields are stored as JavaScript object attributes that are not visible from the Smalltalk side and do not interfere with Smalltalk code.

Detecting Composition Changes without Signatures.

The update header in Figure 7 has to compare composition signatures because a different inlined method might have to be installed even if there are no changes to the layer composition: if a class-specific inlined method is invoked on objects with different layer compositions. Instance-specific inlined method do not have to be invalidated in that case, because every object has its own method (Figure 8).

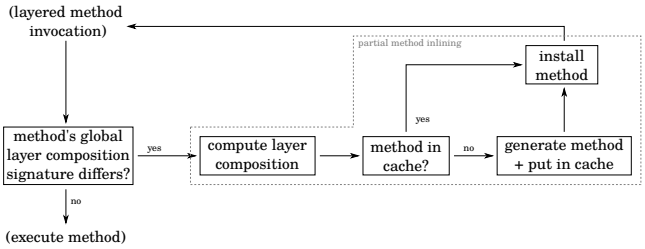


Figure 8: Update header for detecting layer composition changes without comparing layer composition signatures.

If the layer composition is changed for an object, ContextAmber simply removes affected inlined methods from the object. The object's class has corresponding instance methods that will generate and install a new inlined method on the object. No dirty bit is needed.

If the layer composition is changed globally, ContextAmber updates the global layer composition signature for all affected method objects stored in the class' method dictionary, i.e., it updates an instance variable on all affected `CompiledMethod` objects[9]. The update header has the global layer composition signature at the time of inlining hard-coded as a string and compares it with the signature stored in the class' method object. Note, that invariant global layer composition change sequences, i.e., sequences of global layer composition changes that do not change the global layer composition (e.g., activating and deactivating a layer globally), do not invalidate a method, because the global layer composition signature stays the same. ContextAmber uses this technique for instance-specific method inlining.

6. BENCHMARKS

As a benchmark, we are rendering a simplified version of the GhostScript Tiger (Figure 9), consisting of 31 paths and 211 segments in total. If the control point layer is activated, each segment draws at least one control point.

Figure 10 shows the runtime for rendering the tiger 1000 times in a loop[10]. The runtime for the first frame is higher than the average runtime, because this is when an inlined method is created for the first time. The runtime increases if new methods have to be installed: *control point layer (mixed)* denotes the case where the control point layer is active only on 50% of the path objects (on every second one). If method caching is enabled, its class-specific case remains reasonably fast because inlined methods with and without the control point layer are cached; subsequent frame renderings with instance-specific inlining are already fast without caches, because no new methods have to be installed. However, rendering the first frame benefits from method caches even then, because only two methods are then generated and cached (with/without layer). The benchmarks in Figure 10 do not change the layer composition during rendering.

[9]Only layers having a corresponding partial method are part of that signature.

[10]Benchmarks were run on a MacBook Pro with an i7-4558U CPU and 16 GB RAM, using Chrome 41.0.2272.118 (64-bit). We used `git@github.com:matthias-springer/cop-ContextAmber.git` at commit `0006c8b997`.

(a) Without layers. (b) Control point layer.

Figure 9: GhostScript Tiger rendering with Athens.

Figure 11 shows the average runtime for activating and immediately deactivating a layer again. Activating a layer object-wise is slow, because ContextAmber stores layer activation statements in a custom data structure consisting of two stacks: one stack for activations, one stack for deactivations. This is not necessary for global layer activation, because it has the lowest precedence: it is sufficient to store one list of currently activated layers. Instance-specific inlining is slower than class-specific inlining because ContextAmber deletes inlined methods (object) or recomputes the global layer composition signature (global) instead of marking the object dirty or increasing a version number, respectively.

7. RELATED WORK

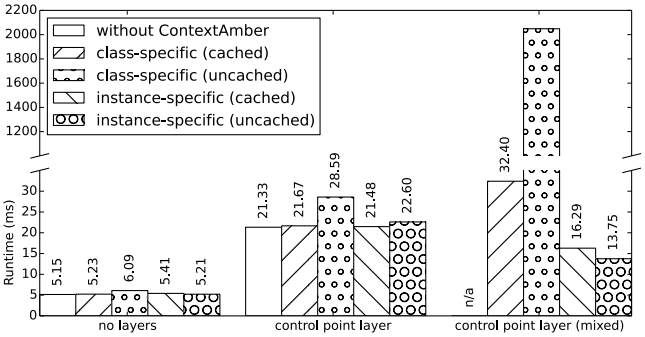
ContextJS is a COP implementation for JavaScript [9]. It relies solely on metaprogramming facilities. It is similar to ContextAmber in a sense that both run on JavaScript. It inlines `proceed` calls in the same way ContextAmber does, and adds an update header to the beginning of every inlined method checking the object's cached layer composition signature; caching is, however, not supported for object-wise layer activation because every object can provide its own method for computing the stack of activated layers [8].

ContextS is a COP implementation for Squeak [5]. Layers are represented by classes and their methods contain the name of the base class, the selector, and the partial method as a block closure in the source code. In ContextAmber, partial classes containing partial methods are associated with layer classes in a many-to-many relationship.

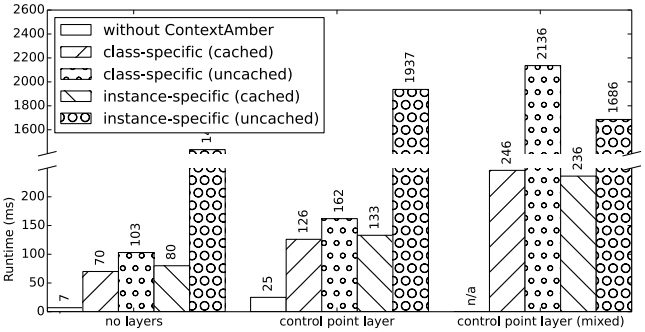
ContextL is a performance-efficient COP implementation for CLOS. Layers are internally represented by classes and layer compositions are cached classes that inherit from (multiple) layer classes: the topmost layer class and a layer composition class for the rest of the layer composition, where the former one takes precendence over the latter one [4].

8. FUTURE WORK

The ideas presented in this work are based entirely on metaprogramming. Changing the language interpreter or virtual machine, however, allows for a variety of different



(a) Avg. frame rendering runtime (excluding first frame).



(b) First frame rendering runtime.

Figure 10: Tiger rendering runtime in milliseconds.

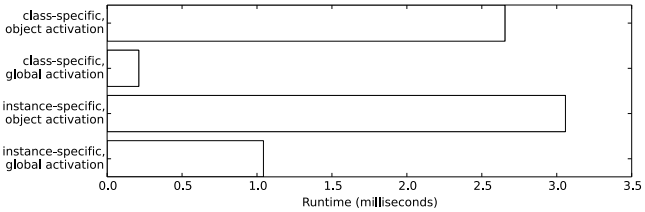


Figure 11: Runtime for activating/deactivating a layer.

implementations. Future work might focus on making polymorphic inline caches [7] aware of layer compositions: the invoked method could be determined based on the polymorphic type of the receiver and its layer composition signature.

Another optimization could make systems, that do already perform aggressive method inlining, aware of layer compositions and `proceed` calls. Basically, `proceed` calls could be treated as a special form of method calls with a different method lookup procedure in the interpreter. Together with special guard clauses, a `proceed` call could look as ordinary to the inliner as any other method call. Partial Evaluation in Truffle [10] is an example of a very aggressive form of method inlining that could be made aware of layer compositions: it continues inlining methods until it encounters a statement that tells it to stop inlining.

Future work could also investigate subclassing of partial classes.

9. SUMMARY

We presented our COP implementation ContextAmber for Amber Smalltalk. ContextAmber reduces the runtime performance overhead of context-oriented programming by caching and executing inlined methods specific to a certain layer composition. ContextAmber can inline methods on a per-instance and on a per-class basis, and depending on the use case, the programmer can choose which one to use. ContextAmber is implemented using metaprogramming facilities, restricting our range of optimizations. Future work might investigate optimizations on the virtual machine level.

10. REFERENCES

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A Comparison of Context-oriented Programming Languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6, New York, NY, USA, 2009. ACM.

[2] P. Coad. Object-oriented Patterns. *Commun. ACM*, 35(9):152–159, Sept. 1992.

[3] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin. Using AspectJ for Component Integration in Middleware. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 339–344, New York, NY, USA, 2003. ACM.

[4] P. Costanza, R. Hirschfeld, and W. De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. In *Proceedings of the 7th Joint Conference on Modular Programming Languages*, JMLC'06, pages 84–103, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] R. Hirschfeld, P. Costanza, and M. Haupt. An Introduction to Context-Oriented Programming with ContextS. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407. Springer Berlin Heidelberg, 2008.

[6] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[7] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[8] R. Krahn, J. Lincke, and R. Hirschfeld. Efficient Layer Activation in Context JS. In *Proceedings of the 2012 10th International Conference on Creating, Connecting and Collaborating Through Computing*, C5 '12, pages 76–83, Washington, DC, USA, 2012. IEEE Computer Society.

[9] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Sci. Comput. Program.*, 76(12):1194–1209, Dec. 2011.

[10] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM.