A Comparison of Context-oriented Programming Languages

Malte Appeltauer Robert Hirschfeld
Michael Haupt Jens Lincke Michael Perscheid
Hasso-Plattner-Institute
University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

ABSTRACT

Context-oriented programming (COP) extensions have been implemented for several languages. Each concrete language design and implementation comes with different variations of the features of the COP paradigm. In this paper, we provide a comparison of eleven COP implementations, discuss their designs, and evaluate their performance.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.2 [**Programming Languages**]: Language Classifications—*Multiparadigm Languages*

Keywords

Context-oriented programming, language comparison

1. INTRODUCTION

The separation of cross-cutting concerns is an issue that is considered by several programming language paradigms, such as aspect-oriented programming [11], feature-oriented programming [3], and context-oriented programming (COP) [10]. The COP paradigm is a relatively novel approach. Since its inception, several COP extensions to various languages—to which we refer as *host languages*—have been developed. Each language implements the core concepts of COP and provides host-language specific functionality. Even though it is apparent that COP is an interesting field for research in programming language design, no systematic comparison of these languages, their design, implementation, or unique features has been done yet.

In this paper, we compare eleven COP languages extending eight host languages. We describe their characteristics and implementation strategies and discuss a performance evaluation of the languages, where we measure the overhead (relative to the respective host language) that is caused by the use of COP constructs.

COP '09 July 7, 2009, Genova, Italy

Section 2 wraps up the COP paradigm and the surveyed languages. Section 3 discusses COP language features and variations. The performance evaluation of the language implementations is presented in Section 4. Section 5 summarizes the paper.

2. COP LANGUAGES

In this section, we give a brief introduction to COP and the language implementations discussed in this paper. For a detailed description of the concepts of COP, we refer to [10]; Section 2.2 provides references for the COP languages considered in this paper.

2.1 Overview

As an extension to object-oriented programming, COP provides means for the concise specification and dynamic composition of behavioral variations that cut across a system. For this purpose, the paradigm introduces the *layer* construct, a modularization concept for behavioral variations that are distributed over several modules, e.g. classes or objects.

Layers contain *partial method definitions* that implement the functionality of a behavioral variation. To distinguish between these methods and common method definitions, we introduce the terms *plain method definition* and *layered method definition*. A plain method denotes a method whose execution is not affected by layers. Layered methods consist of a *base method definition*, which is executed when no active layer provides a corresponding partial method, and at least one partial method definition.

At run-time, layers can be composed for the dynamic extent of a certain control flow. A partial method definition can proceed to a corresponding partial method in another active layer or, if such method does not exist, to the base method definition. The order of layer composition defines the order in which partial methods definitions can be traversed. In general, layers and their partial methods are accessed in the reverse order of their activation: the last activated layer is the first one of the proceed chain.

2.2 Languages

ContextL [4,5] was the first COP extension to a programming language. It is based on Lisp and extends the Common Lisp Object System. Layers can be defined for classes, functions and methods. At run-time, layers can be activated for a certain control flow. ContextL provides several special features that are presented in Section 3.

Subsequently, several meta-level libraries for dynamic pro-

language		ContextL Lisp	ContextS Smalltalk	ContextJ Java	ContextJ* Java	ContextLogicAJ Java	PyContext Python	ContextPy Python	ContextR Ruby	ContextJS JavaScript	ContextG Groovy	cj DelMDSOC
version		0.52	n/a	1.2	1.0	1.0	1.0	1.1	1.0.2	n/a	1.0	n/a
release		02/09	02/09	03/09	04/08	09/08	05/08	11/08	08/08	04/09	02/09	10/08
Language Design												
layer declaration	class-in-layer	x	x				x			x		x
strategy	layer-in-class	x		x	x	x		x	x	x	x	x
layer activation	thread-local		x	x	x	x	x	x	x		x	
	dynamic extent	x	x	x	x		x	x	x	x	x	x
	global	x	(x)[1]	x				x		x		
	explicit deactivation	x	x	x			(x)[1]	x	x	x		x
reflective access	access active layers	x	x	x			(x)[1]	(x)[1]	x	x	x	
	layer introspection	x	x				(x)[1]	(x)[1]		(x)[1]	(x)[1]	
state	access to layers-specific state	x	x				x	x			(x)[1]	x
	introduce state to objects	x		x						x	x	
Implementation												
implementation technique	library / meta-level	x	x		x		x	x	x	x	x	
	pre-compiler / macros	x				x						
	compiler			x								x
	run-time environment											x
layer representation	class	x	x		x	x	x					
	object	x	x				x	x	x	x	x	
	layer type			x								x
layer activation	push to layer stack	x		x	x	x	x	x	x	x	x	
	directly change method lookup		x									x

[1] Functionality only accessible via the host languages meta-level features

Figure 1: Feature comparison of several COP languages.

gramming languages were developed, namely *ContextS* [9] for Smalltalk, *ContextR* [15] for Ruby, *ContextJS* for JavaScript, *ContextPy* [16] and *PyContext* [17] for Python, and *ContextG* for Groovy.

Aside from these dynamic languages, three COP extensions to the Java programming language exist, namely *ContextJ* [1], *ContextJ** [10], and *ContextLogicAJ* [2]. ContextJ* is a Java 5 library that implements the core concepts of COP. The main purpose of the development of ContextLogicAJ was to experiment with various mappings of COP syntax to Java, to find an optimal transformation that can be employed by a compiler-based implementation. It is implemented as an aspect-oriented precompiler based on the *LogicAJ* [12] aspect language and provides a more convenient syntax than ContextJ*. ContextJ is a Java language extension with a dedicated COP syntax. The first ideas about a ContextJ language were presented in [5]; a language specification and compiler-based implementation of ContextJ is provided by [1].

The *cj* prototype [13, 14] is an implementation of a minimal subset of ContextJ. It serves the purpose of demonstrating the applicability of a machine model and semantics for multi-dimensional separation of concerns [8] to COP. The *cj* prototype does not run on a standard Java virtual machine but provides full layer (de)activation capabilities.

In the following sections, we investigate the design and implementation of the COP languages presented so far[1]. Another approach to context-orientation is *Ambience* and its underlying *Ambient Object System* [6] that is based on Common Lisp. It supports behavior adaptations with partial method definitions and context objects, which correspond to COP layers. Due to space limitations, we omit Ambience in our comparison.

[1]Most COP languages are accessible via our COP website at `http://www.hpi.uni-potsdam.de/swa/cop`

3. COP LANGUAGE FEATURES

In the following, we discuss COP language features and the characteristics on which our comparison is based. We distinguish between features of COP language designs and implementation properties of the languages presented in the preceding section. Figure 1 presents an overview of the COP languages and their features.

3.1 Language Design

Layer Declaration Strategy. Layers are the modularization concept for cross-cutting behavioral variations in context-oriented languages. In general, we distinguish two layer declaration strategies, namely *class-in-layer* and *layer-in-class*. Class-in-layer denotes layer declarations where the layer is defined outside the lexical scope of the modules (e. g., classes) for which it provides behavioral variations. Similarly to aspect definitions, this strategy allows layer encapsulation in dedicated modules. This is especially beneficial when programs are evolving: later introduction of layers or layer behavior does not affect the code of already existing modules.

In contrast, layer-in-class supports the declaration of a layer within the lexical scope of the module it augments. The benefit of that strategy is that module definitions are completely specified, which helps code analysis and understandability. Some languages, such as ContextL and ContextCS, allow both strategies and leave it to the programmer to decide whether he wants to define layered methods close to the lexical scope of their default methods, or in other modules.

The semantics of layer-in-class and class-in-layer can slightly differ. In ContextL, for example, layer-in-class allows partial methods to access private elements of its enclosing class, whereas class-in-layer specifications cannot break encapsulation and therefore do not have access to internal state and behavior.

Layer Activation. The key mechanism of COP is the dynamic composition of layers. Several alternatives are possible to control the scope of the activation where the most primitive way is to *globally* activate layers. In most cases, this method is not applicable since it may cause inconsistencies of the system's state. *Thread-based activation* denotes layer (de)activation with thread-local effect. ContextJ and ContextLogicAJ support this control mechanism.

The most common activation strategy provided by COP languages is *dynamic-extent based activation*. COP languages provide a block construct denoting the scope in whose dynamic extent the layer activation is active. Most languages also offer a special construct for dynamic-extent based deactivation.

Some implementations offer global activation across thread boundaries, so that one layer activation concurrently influences several execution paths. This mechanism has to be used carefully since global activation can trigger unintended side-effects.

Reflective Access. Some situations require, besides common layer definition and activation, access to the currently active layers at run-time. A few COP languages offer a reflective API that allows the introspection of the current layer composition. In addition, host languages with strong reflective capabilities allow access to layers and partial methods, though this functionality is not provided via an API but requires the use of internal functions of the COP implementations[2].

Stateful Layers. Besides behavioral variations, some languages offer the introduction of new state via layers. We distinguish between two variants: (1) layers provide state that is accessible in all definitions of a layer (that can be distributed over several classes); and (2) a layer definition introduces state to the classes it augments.

3.2 Implementation Details

The languages investigated in this paper implement layers and dynamic composition in various ways, which are described in this section.

Implementation Technique. The investigated extensions to *dynamic* languages are implemented by means of their host language, i. e., at *library level*. For the extension of dynamic dispatch, the languages make use of their meta-level facilities. The benefit of this implementation technique is quick language development without requiring complex language development environments, such as compiler frameworks and parser generators. Possible drawbacks are performance problems and complex non-declarative syntax.

Other techniques are *precompilers*, such as ContextLogicAJ, and *compilers*, such as ContextJ. Both allow for dedicated syntax and an implementation with good performance. As dynamically updating *compiled* code is not easily achievable in statically compiled languages, these implementations have to maintain data structures capturing layer activation state and providing information for dynamic dispatch.

ContextL is a hybrid approach since it is a meta-level library using the Common Lisp Object System and uses macros supporting a dedicated syntax.

Layer Activation. Two strategies for layer injection can be observed. Layer and partial method definitions can cause the generation of *proxy objects* that are executed instead of the original method when the layer is active. The additional proxy object delegations typically cause an overhead at runtime (see Section 4.1). This variant requires to manage a list of active layers which is consulted (at least) at any method invocation to a layered method.

The other strategy is to dynamically change the delegation chain or virtual method lookup table upon layer activation. In this case, no explicit list of active layers needs to be managed. On the other hand, each layer (de)activation causes the manipulation of all delegations to its partial methods, which might affect many classes.

Layer Representation. By definition, a layer is a language construct orthogonal to classes. COP languages implement layers in different ways. We only consider the layer representation at development time since this is the phase where developers have to deal with layers.

Library-based approaches implement layers in terms of concepts available in their host language—e. g., *classes* or *objects* that provide layer-specific functionality—allowing for more flexibility in the language implementation. For instance, if an application requires an adaption of the COP behavior, this can be easily implemented in the COP library.

Compiler-based languages can extend their host language's syntax and therefore can introduce *dedicated language constructs* for layers and layer activation. Since this language design is not restricted to the known abstractions of the host language, the COP syntax can be more declarative than for library-based languages.

3.3 Special Features

Aside from the general COP functionality, some implementations offer special features for improving and extending COP. We will now give a brief overview of these concepts and their corresponding implementations.

ContextL comes with explicit language support to cope with dependencies between layers. Layers might have functional dependencies on other layers; e. g., a security layer might rely on the presence of a user management layer. To expose and manipulate the run-time layer composition, ContextL provides a comprehensive reflective API supporting *reflective layer activation*. A more declarative *layer dependencies description* is also supported by ContextL.

ContextPy provides the concept of *guards* and the possibility to use COP for procedural programming. The activation of a partial method might not just depend on one layer being active. There might be scenarios in which the activation of a partial method is only possible when a certain combination of layers is active. Guards are functions that receive the list of currently active layers and return a Boolean value indicating whether the partial method this

[2]In Figure 1, we denote the reflective access by means of the host language with *(x)*.

guard was assigned to is to be activated. Moreover, according to its pure decorator implementation, ContextPy does not depend on object-oriented constructs like classes. Thus, COP can be applied to object-oriented as well as procedural programming.

PyContext [17] suggests the concepts of *dynamic variables* [7] and *implicit layer activation*. Dynamic variables allow developers to access contextual state without additional pass-through parameters in the method signature. They are represented as globally accessible objects, whose values are dynamically determined within the dynamic extent of a layer activation. Previous values are hidden and available again after leaving the current extent. For instance, session state can be accessed at any point in time during the execution of the current layer composition. The implicit activation of layers is concerned with the problem of spreading many explicit layer activations over the whole source code. PyContext factors out context activation from the main program logic and defines a method returning whether the layer is active or not. Each time a layered method is called and the layer is registered for implicit activation, the `active` method is executed and its corresponding partial method, if necessary, contributes to the final composition.

In ContextR [15], layer definitions can be extended with so-called *hook methods* that are executed at the beginning and end of a dynamic extent-based layer activation.

4. PERFORMANCE EVALUATION

In this section, we present run-time measurements, based on a set of micro-benchmarks, assessing the overhead caused by layer activation and layer-aware method dispatch compared to only using plain host language features. This analysis is the basis for the ensuing discussion of the different language design and implementation decisions.

The micro-benchmarks were run on a 1.8 GHz dual core Intel Core 2 Duo with 2 GB memory running Windows XP, except the ContextL and ContextJS benchmarks, which were evaluated on a Intel Core 2 Duo, 2.4 GHz, 4 GB, running Mac OS X 10.5.6. We execute the benchmarks with the following versions of the host languages: LispWorks 64-Bit 5.1.2, Java 1.6, Squeak Smalltalk 3.10, Python 2.6, Ruby 1.8.6, JavaScript for Safari 3.2.1, and Groovy 1.6.

4.1 Layer-aware Method Dispatch

In most languages, layer-aware method lookup requires additional operations at run-time that may cause an execution overhead compared to the host language method lookup. To measure the possible overhead, we compared the method execution of plain methods that execute multiple operations with their layered counterparts, where each operation is modularized in a separate partial method. The benchmark class contains ten plain methods (`method_01`–`method_10`) and class variables (`counter_01`–`counter_10`), where method `method_i` increments the fields `counter_01`–`counter_i`. The same behavior is provided by a layered method (`layered`), which only increments `counter_01`, and nine partial method definitions (`pmd_1`–`pmd_9`) of nine distinct layers (`Layer1`–`Layer9`). Each partial method increments a distinct field and proceeds to the next method definition. The call of `method03`, for instance, executes the incrementation of `counter_01`, `counter_02`, and `counter_03`, which is the same behavior as the call to `layered` in the presence of the layers `Layer1` and `Layer2`.

The results are shown in Figure 2. In all languages except ContextL and *cj*, we register a significant performance decrease from 75 % up to 99 %. In those languages that implement COP with meta-programming techniques, we recognize a performance loss of more than 93 %. The two compiler and precompiler based languages, ContextJ and ContextLogicAJ, exhibit the least performance penalties (75–90 %).

The measurement of ContextL contains a broad range of values from 6 % up to 140 % and the results seem not to correlate with the increasing number of counter increments. The anomalies can be explained by massive optimizations provided by LispWorks.

Also, the *cj* prototype performs relatively well; the execution time of a layered method with no active layers even produces no overhead. This is because *cj* directly manipulates the lookup table upon layer activation instead of managing an active-layers list for consultation upon dispatch. The presence of such lists is the main reason for the extreme overheads observed for the other languages.

4.2 Layer Activation

Aside from layer-aware method lookup, we analyzed the costs of dynamic-scope based layer activation, since this is the most common composition mechanism in the considered languages. The activation of a layer requires an update of a set or list of active layers. If the host language supports threads, the active layers must be declared thread-locally. To measure the costs of this update, we compared the execution time of five methods (`method_1`–`method_5`), where `method_i` contains the incrementation of a class variable `counter_i`. For each method, five layers (`Layer1`–`Layer5`) provide a partial method with the same body. Thus, the execution of a method results in the same behavior, independently from the layer composition. We ran `method_1`–`method_5` without active layers, which is the reference value for this benchmark, and with the successive activation of one to five layers.

Figure 3 presents the results of this benchmark. Best results are achieved by ContextL, ContextPy, and ContextJS. The other languages suffer from a significant performance decrease caused by the representation of active layers. Especially for the (pre)compiler-based implementations, layer activation is very expensive.

Worst results are measured for ContextS and *cj*. In contrast to the other implementations, they do not maintain a list of active layers but directly inject and later on remove the code of partial method definitions on each layer activation. This strategy leads for *cj* and also for the first run of ContextS to good results in our first benchmark but causes huge overhead for frequent layer activation.

4.3 Discussion

COP language abstractions, namely layers and dynamic activation, increase the expressiveness of programming languages. However, as our micro-benchmarks of the current versions of some COP languages show, these benefits do not come for free. In most cases the implementations suffer from a performance loss at layer-aware dispatch and layer activations. Some exceptions, such as ContextL and *cj* in the first benchmark and ContextL and ContextPy in the second, significantly reduce the overhead.

At first glance, the results seem discouraging. However, the micro-benchmarks only measure the overhead of COP-

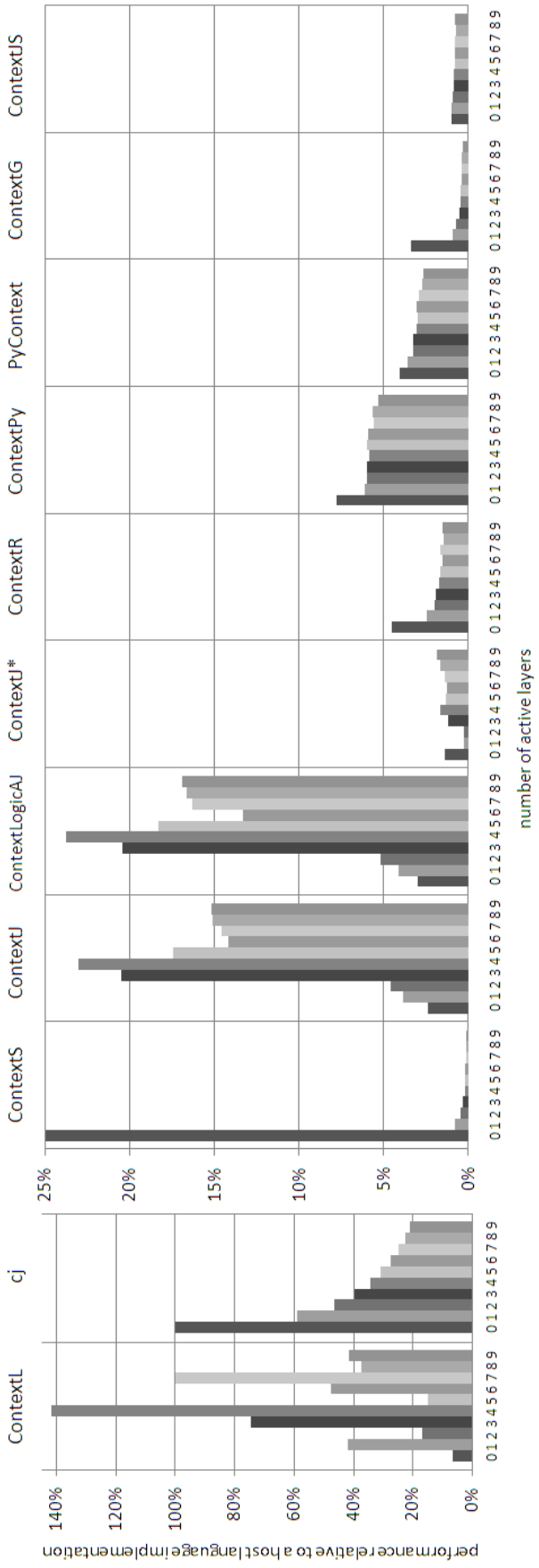

Figure 2: Throughput of layered methods with increasing number of active layers.

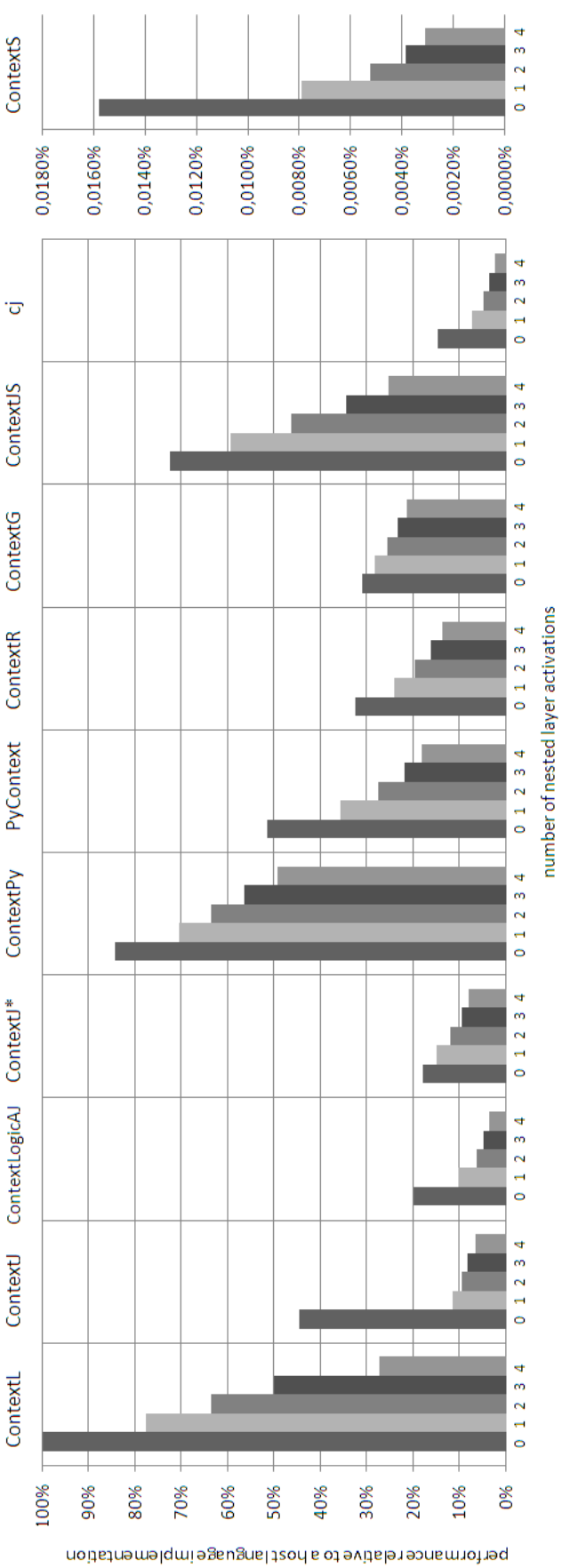

Figure 3: Throughput of zero to four layer activations.

specific code. In a real application, the contingent of this code can be expected to be relatively small, so the observed overheads should be of low relevance.

The observed performance penalties definitely call for the investigation of dedicated optimization strategies in the underlying execution environments. As, e. g., the standard Java VM does not know about the concept of layer-aware method dispatch, it cannot optimize this and other related operations to the same degree it can apply to normal virtual method dispatch. The exceptions mentioned above (i. e., ContextJ, *cj* and ContextPy) suggest directions in which future research should evolve.

5. SUMMARY

The COP paradigm is designed for the separation of context-dependent cross-cutting concerns. It has been implemented in several programming languages, especially for dynamic languages. In this paper, we give an overview on eleven COP languages and present their language design and implementation. The micro-benchmarks in Section 4 show that the current language implementations suffer from a big execution overhead. Future work should definitely consider performance issues of the COP languages, which is a vital property for larger applications.

Acknowledgments

We are grateful to Pascal Costanza for valuable discussions and his help on the ContextL benchmarks. We thank Gregor Schmidt for the implementation of the ContextR benchmarks and Arvid Heise for insights into ContextG. LispWorks Ltd. kindly provided an evaluation license of LispWorks® 64-bit for Macintosh for testing purposes.

6. REFERENCES

[1] M. Appeltauer. ContextJ – Context-oriented Programming for Java. In *Proceedings of the 3rd Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering*, number 27. Hasso-Plattner-Institut, Potsdam, Germany, 2009.

[2] M. Appeltauer, R. Hirschfeld, and T. Rho. Dedicated Programming Support for Context-aware Ubiquitous Applications. In *UBICOMM 2008: Proceedings of the 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 38–43, Washington, DC, USA, 2008. IEEE Computer Society Press.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2003.

[4] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[5] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In D. E. Lightfoot and C. A. Szyperski, editors, *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103, Berlin, Heidelberg, Germany, September 19 2006. Springer-Verlag.

[6] S. González, K. Mens, and A. Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

[7] D. R. Hanson and T. A. Proebsting. Dynamic variables. *SIGPLAN Notices*, 36(5):264–273, 2001.

[8] M. Haupt and H. Schippers. A Machine Model for Aspect-Oriented Programming. In E. Ernst, editor, *21st European Conference on Object-Oriented Programming, ECOOP 2007*, volume 4609 of *Lecture Notes in Computer Science*, pages 501–524, Berlin, Heidelberg, Germany, August 2007. Springer-Verlag.

[9] R. Hirschfeld, P. Costanza, and M. Haupt. An Introduction to Context-Oriented Programming with ContextS. In J. S. Ralf Lämmel, Joost Visser, editor, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7. 2007, Revised Papers*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407, Berlin, Heidelberg, Germany, 2008. Springer-Verlag.

[10] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *Proceedings 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[12] G. Kniesel, T. Rho, and S. Hanenberg. Evolvable Pattern Implementations need Generic Aspects. research report C-196, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo, Japan, June 2004.

[13] H. Schippers, M. Haupt, R. Hirschfeld, and D. Janssens. An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns. In *Proc. SAC PSC*. ACM Press, to appear, 2009.

[14] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. *SIGPLAN Not.*, 43(10):525–542, 2008.

[15] G. Schmidt. ContextR & ContextWiki. Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[16] C. Schubert. ContextPy & PyDCL - Dynamic Contract Layers for Python. Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[17] M. von Löwis, M. Denker, and O. Nierstrasz. Context-oriented Programming: Beyond Layers. In S. Demeyer and J.-F. Perrot, editors, *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, volume 286 of *ACM International Conference Proceeding Series*, pages 143–156, New York, NY, USA, 2007. ACM Press.