MODULARITY'16: G:

Language Oriented Modularity: A Practical Approach *

Arik Hadas
Open University, Raanana 43107, Israel
arik.hadas@openu.ac.il

Abstract

Language Oriented Modularity (LOM) is a methodology that involves the implementation and immediate utilization of domain-specific languages (DSLs) that are also aspect-oriented (DSALs). In principle, DSALs are often the right tool for improving the modularity of modern software projects and an LOM process is necessary due to the difficulty to reuse them across applications. In practice, however, LOM is often impractical and DSALs are underutilized in real-world projects. In this work, we present a novel approach to reduce the implementation cost of DSALs and increase the effectiveness of using DSALs. Consequently, the cost-effectiveness of LOM is improved to a level that makes it practical for a real-world software development process. We validate our approach by implementing it concretely for Java. We evaluate our approach by examining the cost-effectiveness of applying LOM using our approach to open source projects.

1. Introduction, Motivation, and Problem

Language Oriented Modularity (LOM) [13, 18], taking after *Language Oriented Programming (LOP)* [3, 22], is a programming methodology that promotes *on-demand* development and use of *Domain Specific Aspect Languages (DSALs)* during the software modularization process. A DSAL [5] is a programming language that is both domain-specific and aspect-oriented. It provides not only domain-specific abstractions and notations like an ordinary *Domain Specific Language (DSL)* does, but also a modularization mechanism for the separation of domain-specific crosscutting concerns.

With LOM, DSALs are tailored to the crosscutting problems at hand, rather than force-fitting the latter to the available *General Purpose Aspect Language (GPAL)*. The LOM process evolves middle-out. One starts with defining DSALs that best suit the application-specific modularization needs and then works outwards, combining high level programming with these DSALs in parallel to their low level implementation. Since DSALs have only a slim prospect of being reused across applications, a viable LOM process for their on-demand development is essential.

In pursuing LOM practicality, LOP is our baseline for comparing cost-effectiveness. In LOP, the cost of DSLs is low thanks to the use of language workbenches. Developing a new DSL with a language workbench amounts to writing a transformer (generator) from that DSL to a *General Purpose Language (GPL)*, and writing a transformer is much easier than writing a compiler or an interpreter. A language workbench also provides tool support for implementing the transformation and for effective editing of DSL code. Once the DSL code is transformed, it is compiled with the GPL's compiler into an executable form, allowing all development tools that are available for the GPL to be used effectively.

In comparison, the language development cost for DSALs remains high and aspect development tools typically break on DSAL code [7, 8].

1.1 High Development Cost

Obliviousness has traditionally been a long-standing principle in AOP [6]. However, in the context of code transformations, complete obliviousness is disadvantageous. Consider AspectJ as the target language for implementing several DSALs. In AspectJ, the base code cannot refuse advisement (prevent join points from being advised). Consequently, a code transformation that does not preserve the join point "fingerprint" of the original code is not necessarily semantic-preserving in the presence of foreign aspect code. Indeed, translating aspects from different DSALs into aspects in AspectJ and compiling them with the AspectJ compiler (`ajc`) may yield incorrect behavior [19]. When the implementation of DSALs via transformation to a GPAL does not work in general, language workbenches are of little use.

Another difficulty is weaving pieces of advice written in different DSALs at the same join point shadow. A conflict occurs when the various pieces of advice are woven in the wrong order. AspectJ provides some control over the ordering of advice by declaring precedence between aspects (via the `declare precedence` statement). However, for programming with multiple DSALs one may need a finer grained ordering mechanism.

* This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 1440/14.

Tool Support	CF	LW+ GPAL	LW+ CF	Practical LOM
DSAL interoperability	✔		✔	✔
Development process		✔		✔
Editing tools		✔	✔	✔
Aspect tools				✔
Compilation				✔

Table 1. Comparison of tool support for LOM

1.2 Low Effectiveness

Development tools for aspect languages heavily rely on the representation of advice-join-point relationships in order to annotate the source code with hints on how aspects are to be woven into the base code. However, code that is generated from DSAL code loses track of the location of advice in the original DSAL code. Consequently, development tools cannot annotate the code, thus hindering effective programming with the DSAL.

In addition, the ability to compile the software from the command line is of a particular interest in real-world projects because of the use of modern tools for continuous integration and continuous delivery. The fact that `ajc` cannot take DSAL code as input, requires one to modify the compilation process significantly in order to compile the software from the command line.

1.3 Problem in a Nutshell

DSALs seem to be "second-class" [7]. DSALs are more costly to produce than ordinary DSLs and less effective to program with than GPALs (relative to DSLs *w.r.t.* GPLs). On the one hand, DSALs are second-class DSLs because aspect code transformation introduces a *semantic gap* that renders their implementation with existing language workbenches incorrect in the presence of other DSALs. On the other hand, DSALs are second-class aspect languages because aspect code transformation also introduces an *abstraction gap* that renders existing GPAL development tools ineffective for programming with DSALs.

The challenge of pushing the cost-effectiveness of LOM to the baseline of LOP is twofold. First, the semantic gap needs to be bridged to allow language designers implement composable DSALs using language workbenches. Second, the abstraction gap needs to be bridged to provide end-programmers with DSAL development tools that are on par with those available for a GPAL. This includes not only aspect development tools and build tools that typically break on DSAL code, but also general editing tools, such as text-highlighting and auto-completion that DSALs typically lack.

2. Background and Related Work

DSALs are used for many and various crosscutting problem domains [5]. Their domain-specific grammar makes them easier to use by domain experts. Their ability to encapsulate in aspect modules crosscutting logic, which is otherwise scattered and tangled in the application's base code, improves software modularity. The code within these modules is woven into the application's base code. The process of matching advice to join points and weaving the advice accordingly is done by an *aspect weaver*.

The specification of where and how advice should be woven typically makes aspects tightly coupled with the structure of the application code. Moreover, aspects typically need to retrieve data from the base code. This makes them tightly coupled also with the representation of the data in the application code. In contrast, ordinary DSLs are typically neither coupled with the structure nor with the data representation in the application, because the application decides when to invoke DSL code and to provide it the required data. Therefore, DSALs tend to be more coupled with the application they were designed for than ordinary DSLs are.

Several approaches have been proposed to reduce the coupling between aspects and base code. For instance, by using *join point interfaces* [2] the base code can initiate the call to the aspect and provide it the required data, similar to the manner applications interact with ordinary DSLs. However, these approaches have not made it into AspectJ.

Various development tools aim at facilitating the development of DSALs. The Aspect Bench Compiler (`abc`) [1] is more extensible than the AspectJ compiler (`ajc`), allowing one to produce extensions to AspectJ and DSALs more easily. However, `abc` is intended for the development of a particular extension rather than for the composition of extensions and therefore is not compatible with LOM.

Table 1 depicts a comparison of existing tools that could be considered for applying LOM. *Aspect Composition Frameworks (CF)* are designed for the composition of DSALs. Some composition frameworks propose transformation approaches. In XAspects [20], DSALs are transformed into AspectJ. In Reflex [21], DSALs are transformed to a low-level kernel language. However, DSALs implemented using these approaches would not be provided with supportive tools for end-programmers. Moreover, these transformation approaches could lead to wrong program behavior (e.g., deadlock) when multi-DSAL conflicts [15] occur.

Multi-DSAL conflicts are conflicts that may happen when multiple aspect languages are being used simultaneously. They can be classified as foreign advising conflicts and co-advising conflicts [19]. Foreign advising refers to a scenario in which an aspect written in one DSAL advises a join point that resides within a foreign aspect written in a different DSAL. A conflict may happen when that join point is not supposed to be advised. Co-advising refers to a scenario in which two aspects written in different DSALs advise the same join point in the base code. A conflict may happen when the aspects are applied (woven) in the wrong order. The AWESOME composition framework [16] was designed for handling multi-DSAL conflicts. It generalizes `ajc`. Each

DSAL is implemented as a plugin. Rules are set for preventing foreign- and co-advising conflicts. This way one can produce composable DSALs for LOM. However, it requires one to implement a compiler (weaver) plugin, which is a complex task for most programmers. This task is not needed for ordinary DSLs and thus is not supported by existing language workbenches.

Alternatively, one can use a language workbench (LW) and transform DSALs into an existing GPAL. That way, the DSAL development process becomes similar to that of ordinary DSLs. Additionally, general editing tools can be easily generated for the DSALs. However, such a transformation is conceptually similar to that done in XAspects and thus can be ruled out for LOM due to multi-DSAL conflicts.

Lastly, a naive composition of a language workbench and a composition framework simplifies DSAL development (including handling of multi-DSAL conflicts) and enables to generate general editing tools for the DSAL. However, the development cost remains high because of the use of a composition framework and the effectiveness remains low due to the lack of aspect development tools and build tools [7].

Elsewhere [9] we presented an improved approach for the composition of a language workbench and a composition framework that is able to produce DSALs that regain their first-class status with respect to aspect languages. However, DSALs remain second-class DSLs and since the development cost is of a significant importance in the context of the LOM process, this solution is not suitable for LOM.

3. Approach and Uniqueness

To make LOM practical we present an approach that enables the efficient implementation of DSALs by transformation to a GPAL. With our approach DSAL code is transformed to GPAL code *annotated with metadata*, where the metadata is used to bridge the semantic gap and the abstraction gap, thus granting DSALs first-class status, in which the DSAL implementer can leverage exiting language workbenches and the DSAL end-programmer can leverage existing development tools. This provides an end-to-end solution to applying LOM in practice.

Specifically, the metadata is in the form of Java annotations and an interface for invoking transformations. We use annotations to weaken AspectJ in terms of obliviousness, strengthen it in terms of advice ordering, and enhance it in terms of bridging source code locations.

To bridge the semantic gap, a subset of the annotations control the visibility of join points, thus allowing the definition of the transformation to specify where to suppress join point shadows (foreign advising). Another annotation controls the order in which pieces of advice from different DSALs are activated at the same join point shadow (co-advising).

To bridge the abstraction gap, metadata can be attached within an annotation, enabling AJDT to provide first-class browsing and navigation capabilities for DSALs. Additionally, DSAL code transformation plugins that implement a special interface are invoked automatically by the compiler in order to provide first-class compilation for DSALs.

3.1 First-Class DSLs

The key for making DSALs first-class DSLs is handling multi-DSALs conflicts declaratively by setting metadata during the code transformation. By attaching metadata to a particular program element in the generated code, one can specify which join points that are associated with that program elements should be suppressed from foreign aspects. This process prevents foreign advising conflicts. By attaching metadata to an advice in the generated code, one can specify an ordering value that would allow advice-level ordering (rather than aspect-level ordering). This granularity of advice ordering prevents co-advising conflicts.

This reduces the cost of implementing DSALs to that of ordinary DSLs. The implementation process of DSALs then amounts to parsing the domain-specific syntax and transforming it into an existing language. No additional adjustments, such as modifying the compiler, are required. That makes the implementation process of DSALs similar to that of ordinary DSLs that even existing language workbenches for ordinary DSLs can be used for implementing DSALs.

3.2 First-Class Aspect Languages

The key for making DSALs first-class aspect languages is having the GPAL compiler treat DSAL code as if it were GPAL code in terms of its external API. By attaching metadata to advice in the generated code that preserves its source location in the original DSAL code, the compiler can represent advice-join-point relationships in a similar way to their representation with GPAL code. By enabling the compiler to invoke the code transformation internally as a standalone utility, the compiler can receive DSAL code as an input.

This increases the effectiveness of programming with DSALs to the level of that of GPALs. By pointing to the original DSAL code within advice-join-point relationships, aspect development tools for the GPAL can work properly with DSAL code and provide the browsing and navigation capabilities that are typically available while programming with a GPAL. By enabling the compiler to receive DSAL code as an input, build tools for the GPAL can work properly with DSAL code, allowing to compile the project as if the DSAL code were GPAL code. Finally, by implementing the DSAL using a language workbench, one can generate an IDE plugin that provides common editing tools for the DSAL from within the language workbench. This completes the development tools that are generally available while programming with a GPAL.

3.3 Uniqueness

The uniqueness of our approach is twofold. First, our approach is “wholistic”, taking into consideration the whole

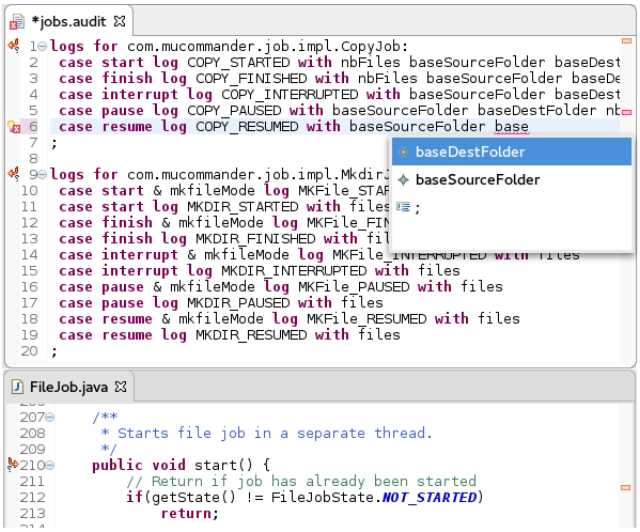

Figure 1. DSAL (top) and base (bottom) code in Eclipse.

LOM process. Unlike alternative approaches that target only certain elements in the implementation or use of DSALs, our approach considers the end-to-end LOM process as a whole, including all the stakeholders involved.

Second, our approach is designed with practicality in mind. One could argue for a finer-grained constructs for handling multi-DSAL conflicts (e.g., those provided by AWESOME) or for support of a wider range of DSALs (e.g., those that are not reducible to a GPAL). However, that would likely be at the expense of the cost-effectiveness, and thus the practicality, of LOM. Our approach provides a process and tools for addressing an important family of crosscutting concerns that can be found in real-world projects. Possible extensions to our approach that do not compromise the practicality of LOM is left for future work.

4. Results, Contribution, and Conclusion

To validate our appraoch we extended AspectJ with a small set of annotations and an interface. First, `@Hide` annotations enable to specify which join points associated with a program element (type, method, or field) to suppress from other aspects. Second, the `@Order` annotation enables to specify an ordering value for an advice. Third, the `@BridgedSourceLocation` annotation can store the source code location of advice in the original DSAL code. Finally, a `Transformation` interface enables the compiler to invoke code transformations internally.

We modified `ajc` in order to support these extensions.[1] Our modifications to `ajc` are *optional* — when not in use, the compiler's behavior is unaffected, thus preserving the correctness of the weaving in `ajc` before the change — and *minimal* — we do the minimal changes necessary to support our extensions, thus we expect the process of reapplying these changes to a newer version of the compiler to be relatively straightforward.

Implementation	Grammar	Code Transformation		Weaver Plugin
		EV	Other	
Language	**SDF**	**Stratego (AST)**		**Java**
CF Approach	34	761 (4168)	297 (3001)	1557
Our Approach	34	0	382 (3008)	0

Table 2. LOC in two implementations of COOL

To evaluate the practicality of our approach in various scenarios we conducted three case studies, two experimental and one comparative.

In the first case study, we implemented a DSAL for a new crosscutting feature of auditing and used it to program aspect solutions for file operations in the muCommander project.[2] This process, that represents a typical scenario for LOM, demonstrates the improved cost-effectiveness of LOM achieved with our approach. The implementation of the DSAL was completed entirely using the Xtext language workbench [4], just like implementing an ordinary DSL. We were able to use development tools while programming with the DSAL. Figure 1 demonstrates some of these tools. Auto-completion and syntax-error checking (line 6) and text-highlighting for the DSAL code are shown. In addition, AJDT markers are displayed next to both the DSAL code (line 9) and the base code (line 210).

In the second case study, we applied LOM to implement three DSALs for crosscutting concerns found in the oVirt project[3] [10]. Using these DSALs we managed to separate out the mentioned concerns. Code scattering was eliminated by encapsulating code that was spread across many classes in a single module implemented in the corresponding DSAL. Code tangling was resolved by extracting code that was tangled in a common root class into the DSAL aspects. This case study provided an additional evidence for the improved cost-effectiveness of LOM using our approach. First, each DSAL was implemented completely in Xtext in just a few hours. Second, we were provided with the desired editing tools, aspect development tools, and build tools, that enable effective programming with the DSAL even in a complex and large-scale project like oVirt.

Finally, we implemented COOL [17], a complex DSAL often used as a benchmark example. The implementation of COOL provided yet another evidence for the reduced implementation cost using our approach. This time the implementation was done completely using the Spoofax language workbench [14], demonstrating that our approach is agnostic to the selection of the language workbench. The synchronization of a bounded-stack using a coordinator (aspect) in

[1] The code changes made to `ajc` and those that were done as part of the case studies are available at `https://github.com/OpenUniversity/`.

[2] `http://www.mucommander.com/`

[3] `https://www.ovirt.org/`

COOL provided yet another evidence for the effective programming with supportive tools. It also illustrated the effectiveness of our extensions to AspectJ in handling multi-DSAL conflicts. The deadlock problem reported elsewhere [15] that occurs when the coordinator is translated to plain AspectJ was not observed when the `@Hide` annotations were placed during the transformation (but reproduced successfully when we removed them). However, the most interesting finding in this case study was the results of comparing our implementation with that in the AWESOME composition framework. Table 2 shows that our approach required much less code and the required knowledge was in higher-level technology (AspectJ rather than bytecode manipulation).

The main contribution of this work is a practical approach that addresses the Achilles' heel of LOM, namely that the DSAL development process is far from being cost-effective. Our approach is capable of producing DSALs by applying the same process and tools used for ordinary DSL development. This brings the cost-effectiveness of LOM closer to the baseline of LOP, and makes LOM practical for a real-world software development process [12, 13].

We also contribute case studies on the effectiveness of application-specific DSALs in handling crosscutting features that are found in real-world projects. The relative ease of implementing them with LOM makes even their one-time use cost-effective [11]. More broadly, thinking of aspect languages as application-specific or even disposable languages brings about a more agile-like process in designing and using DSALs. It might also suggest that future research should focus on making GPALs more expressive (by exposing more join points, for example) rather than attempting to make DSALs more reusable.

References

[1] P. Avgustinov et al. abc: an extensible AspectJ compiler. In *AOSD'05*, pages 87–98, Chicago, Illinois, USA, Mar. 2005. ACM Press.

[2] E. Bodden and E. Tanter. Join point interfaces for safe and flexible decoupling of aspects. *ACM Softw. Eng. Methodol.*, 23(1):7:1–7:41, Feb. 2014.

[3] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), 2004.

[4] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Eclipse Modeling Symposium, Eclipse Summit Europe 2006*, Esslingen, Germany, Oct. 2006.

[5] J. Fabry, T. Dinkelaker, J. Noyé, and E. Tanter. A taxonomy of domain-specific aspect languages. *ACM Computing Surveys (CSUR)*, 47(3), Apr. 2015.

[6] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns*, 2000.

[7] A. Hadas and D. H. Lorenz. Demanding first-class equality for domain specific aspect languages. In *Modularity'15 Comp.*, pages 35–38, Fort Collins, CO, USA, Mar. 2015. ACM Press. Position paper.

[8] A. Hadas and D. H. Lorenz. First class domain specific aspect languages. In *Modularity'15 Comp.*, pages 29–30, Fort Collins, CO, USA, Mar. 2015. ACM Press. Poster Session.

[9] A. Hadas and D. H. Lorenz. A language workbench for implementing your favorite extension to AspectJ. In *Modularity'15 Comp.*, pages 19–20, Fort Collins, CO, USA, Mar. 2015. ACM Press. Demo Session.

[10] A. Hadas and D. H. Lorenz. Application-specific language-oriented modularity: A case study of the oVirt project. In *MASS'16*, pages 178–183, Málaga, Spain, Mar. 2016. ACM Press.

[11] A. Hadas and D. H. Lorenz. Toward disposable domain-specific aspect languages. In *FOAL'16*, pages 83–85, Málaga, Spain, Mar. 2016. ACM Press.

[12] A. Hadas and D. H. Lorenz. Toward practical language oriented modularity. In *LaMOD'16*, pages 94–98, Málaga, Spain, Mar. 2016. ACM Press.

[13] A. Hadas and D. H. Lorenz. Language oriented modularity: From theory to practice. *The Art, Science, and Engineering of Programming*, 1(10):1–37, Apr. 2017.

[14] L. C. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *SPLASH'10*, pages 444–463, Reno/Tahoe, Nevada, USA, Oct. 2010. ACM.

[15] S. Kojarski and D. H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In *OOPSLA'05*, pages 247–263, San Diego, CA, USA, Oct. 2005. ACM Press. ISBN 1-59593-031-0.

[16] S. Kojarski and D. H. Lorenz. Awesome: An aspect co-weaving system for composing multiple aspect-oriented extensions. In *OOPSLA'07*, pages 515–534, Montreal, Canada, Oct. 2007. ACM Press.

[17] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997.

[18] D. H. Lorenz. Language-oriented modularity through Awesome DSALs: summary of invited talk. In *DSAL'12*, pages 1–2, Potsdam, Germany, Mar. 2012. ACM Press.

[19] D. H. Lorenz and S. Kojarski. Understanding aspect interactions, co-advising and foreign advising. In *Proceedings of ECOOP'07 Second International Workshop on Aspects, Dependencies and Interactions*, pages 23–28, Berlin, Germany, July 2007.

[20] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain specific aspect languages. In *OOPSLA'03 Comp.*, pages 28–37, Anaheim, California, 2003. ACM Press.

[21] É. Tanter. Aspects of composition in the Reflex AOP kernel. In *SC'06*, Vienna, Austria, Mar. 2006.

[22] M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.