Toward Practical Language Oriented Modularity [*]

Arik Hadas[1] David H. Lorenz[1,2 †]

[1]Open University, Raanana 43107, Israel
[2]Technion–Israel Institute of Technology
Haifa 32000, Israel
arik.hadas@openu.ac.il, dhlorenz@cs.technion.ac.il

Abstract

Language oriented modularity (LOM) implies the use of multiple domain specific aspect languages (DSALs) simultaneously. The complexity of implementing these DSALs has significant implications on the practicality of LOM. In this paper we describe a transformation-based approach to DSAL implementation that reduces the overall cost of implementing the weaving semantics and thus improves the practicality of LOM.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures; D.2.6 [*Software Engineering*]: Programming Environments—Programmer workbench.

General Terms Language, Design.

Keywords Aspect oriented programming (AOP), Domain specific aspect language (DSAL), Domain specific language (DSL). Language oriented modularity (LOM).

1. Introduction

Language Oriented Modularity (LOM) [16] is a methodology that puts *Domain Specific Aspect Languages (DSALs)* [5] at the center of the *Aspect Oriented Software Development (AOSD)* [6] process. It involves the development and use of DSALs on-demand during the software modularization process. LOM is a specialization of *Language Oriented Programming (LOP)* [4, 21], applied to DSALs rather than *Domain Specific Languages (DSLs)*.

From a language definition perspective, DSALs resemble DSLs. Both define a programming language with domain specific notations and abstractions aiming at simplifying problems of a particular domain and scope. However, defining a DSAL is slightly more complex than defining a DSL due to the need to also specify the weaving semantics. In particular, the definition of a DSAL must specify how aspects written in other languages may advise code written in the DSAL (foreign advising) and how aspects written in the various DSALs may collaboratively affect the same join point (co-advising) [15, 17].

From a language implementation perspective, DSALs and DSLs differ greatly. To help with the development of DSLs, LOP typically utilizes *language workbenches* [7], whereas LOM lacks similar tool support for DSALs. While language workbenches can assist in parsing DSAL programs, they do not help with implementing the weaving semantics. Implementing a weaver often requires expertise in low-level programming, which further increases the cost of DSAL development.

From a language use perspective, DSALs are like *General Purpose Aspect Language (GPALs)*, since both are used to modularize crosscutting concerns. However, it is more complicated to use a DSAL than a GPAL. On the one hand, the specificity of a DSAL to a particular problem domain decreases the programming effort. On the other hand, lack of supportive development tools and build tools for DSALs increases the programming effort. Although GPALs typically come with a suite of development tools that facilitate aspect development, these tools are tailored to a specific GPAL and do not work with DSAL code properly. In addition, the build process might need to be modified in order to support DSAL code.

2. Working Hypothesis

Today, developers often choose to modularize crosscutting concerns directly with GPALs rather than through LOM with DSALs, or worse yet, not to separate them out at all. We believe the reason for this is twofold. First, the implementation of a dedicated weaver (or weaver plugin) for a DSAL requires low-level programming, a difficult task for most programmers. Second, custom-made DSAL weavers make programming more difficult because development tools that work with GPALs break on DSAL code.

Our working hypothesis assumes that making DSALs more like DSLs, in terms of language definition and implementation, and more like GPALs, in terms of language use, could significantly improve the cost-effectiveness of the LOM development process to the extent that the LOM methodology may become practical for real-world software development.

More Like DSLs DSLs are relatively straightforward to define and implement thanks to the use of language workbenches. To become more like DSLs, the DSAL development process should be backed up with supportive development tools that provide a similar language design experience.

Implementing a DSL in a language workbench basically amounts to a transformation of the DSL to a general-purpose language (GPL). Language workbenches even provide the language designer with a DSL for specifying the code transformation, along with IDE tools for supporting the development of the transformation. What additionally simplifies the implementation of the targeted DSL is the

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

[†] Work done in part while visiting the Faculty of Computer Science, Technion—Israel Institute of Technology.

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...
http://dx.doi.org/10.1145/2892664.2892678

ability to use the build tools of the GPL instead of developing dedicated build tools for the DSL.

In contrast, implementing a DSAL cannot be carried out by means of a simple transformation to a GPAL. A naive transformation generally does not preserve the weaving semantics and may even result in incorrect behavior in the context of simultaneous use of multiple DSALs [13]. Therefore, one often resorts to other less supported aspect composition frameworks for implementing the weaving semantics.

More Like GPALs Compared to GPALs, DSALs have superb declarative syntax but lack supportive development tools for programming in it. To become more like GPALs, the programming experience with DSALs should be raised to the level one is normally used to when programming with GPALs.

GPAL programmers have at their disposal IDEs with editing tools, such as auto-completion and syntax highlighting, and aspect development tools, such as AJDT [3] (for AspectJ) that displays which join points are affected by which advice and vice versa.

In contrast, DSAL programmers rarely have a development environment available to them. GPAL development tools do not work on DSAL code, because they rely on a certain internal representation of the metadata. A custom-made weaver (plugin) for a DSAL typically does not represent the relationship between advice and join points in the same way that these tools expect.

3. Approach

We present a variant of LOM which is restricted to tackling crosscutting concerns that in principle could have been modularized directly with a GPAL. Under this restriction, the DSALs that are developed during the LOM process can be transformed to a GPAL-based kernel language, thus reducing the effort needed to design, implement, or use them. This in turn makes this variant of LOM practical.

3.1 Transformation to a GPAL-based Kernel Language

In our approach, the comprehensive set of high-level constructs that are available in the GPAL can be used for implementing the weaving semantics of the DSAL. This significantly reduces the cost of implementing the DSAL to the extent that it promotes DSALs that are tailored to the problem even at the expense of being less reusable. When the expectation for reuse is lower, DSALs can provide less constructs and these constructs can be made even more specific to the problem at hand. This in turn can simplify even further the DSAL definition and its implementation.

During the transformation of the DSAL code to the kernel language, elements within the transformed code are annotated with their original location in the DSAL code. The weaver of the (kernel) GPAL is modified to refer to the annotated rather than the actual location. This way, aspect development tools that work with the GPAL provide meaningful cross-reference information when programming in the DSAL.

3.2 Multi-DSAL Conflict Resolution

In order to handle foreign advising, the transformation of DSAL code to the kernel language annotates in the transformed code join point shadows that should be hidden from other DSALs, thereby indicating to the weaver not to expose them. In order to handle co-advising, the transformation also annotates advice in a way that they can be identified by a predefined comparator[1] and the comparator is extended in order to sort these advice.

[1] `http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html`

To avoid a combinatorial explosion, the weaver provides a default resolution for both foreign advising and co-advising. For foreign advising the default is to expose all join point shadows defined within the DSAL code to all other DSALs. For co-advising the default is to weave advice according to the order the DSALs were composed. The default resolution can be overridden either by specifying for a particular DSAL the join point shadows that need to be hidden or by defining an explicit order for particular pieces of advice.

3.3 Leveraging a Language Workbench

By transforming DSALs to a kernel language and defining the resolution for foreign advising and co-advising as part of this transformation, the development of a DSAL becomes closer to that of a DSL and most of the development can be done using a traditional language workbench. The language definition can be done by defining the grammar of the DSAL using tools that language workbenches provide for defining DSLs. The language implementation can be done by transforming DSAL code into the kernel language using tools that language workbenches provide for transforming DSLs. The only thing that cannot be done in the language workbench is the implementation of the co-advising comparator.

Thanks to the use of a language workbench, not only is the cost for defining and implementing the DSAL reduced, but also the cost of using that DSAL is reduced because the language workbench can generate editing tools for programming in the DSAL.

4. Demonstration

We demonstrate our approach by describing the language design, implementation, and use of a DSAL we developed for a crosscutting concern found in the oVirt project.[2] We apply the approach with AspectJ as the GPAL on which the kernel language is based. We chose AspectJ because oVirt is implemented in Java. However, the approach is applicable also to GPALs other than AspectJ.

4.1 About oVirt

oVirt is an open-source enterprise application for providing and managing virtual data centers and private cloud solutions.

oVirt-Engine is the control center of the oVirt distributed system that manages the different hosts that run virtual machines (VMs). It is a Java server application that serves as the front-end of the entire data center, executing operations it receives from clients and reporting back to them the up-to-date status of the data center.

The core design of oVirt-Engine is based on the COMMAND design pattern [8]. Each operation that is supported by oVirt-Engine is modeled by a command class that inherits from a common root called `CommandBase`.

4.2 Synchronization in oVirt-Engine

We have found several concerns in oVirt-Engine that crosscut many modules in the oVirt-Engine application. One of those concerns is about preventing conflicting commands from running simultaneously. For that, each command defines its read-locks (the method `getSharedLocks`), its write-locks (the method `getExclusiveLocks`) and global properties for its locks (the method `applyLockProperties`). Each lock includes the type and identifier of the entity being locked along with a message that is displayed once this lock prevents another command from being executed. The global properties include the scope of the locks (defines whether the locks should be released at the end of the synchronous or at the end of the asynchronous execution of the command), and whether the command should wait until it manages to acquire the required locks or to fail if they cannot be acquired.

[2] `http://www.ovirt.org`

Listing 1: Locks.xtext

```
Model: (commands+=Command)*;

Command:
  'locks for' type=[types::JvmDeclaredType|QualifiedName] '('
        scope=Scope (wait?=('& wait'))? ')' ':'
    (exclusiveLocks=Exclusive)?
    (sharedLocks=Inclusive)?
    (message=Message)?
    ';'
;

enum Scope: sync|async;

Exclusive:
  {Exclusive}
  'exclusively' (override?='(overrides)')? '{'
      (locks+=Lock(',' locks+=Lock)*)?
  '}'
;

Inclusive:
  {Inclusive}
  'inclusively' (override?='(overrides)')? '{'
      (locks+=Lock(',' locks+=Lock)*)?
  '}'
;

Lock: 'group: ' group=[types::JvmEnumerationLiteral] 'instance: '
      id=[types::JvmOperation] (conditional?='if' condition=[
      types::JvmOperation])?;

Message: 'message: ' type=[types::JvmEnumerationLiteral] (vars
      +=Var)*;

Var: '<' key=STRING ',' value=[types::JvmOperation] '>';

QualifiedName: ID ("." ID)*;
```

4.3 *oVirtSync*

We implemented a DSAL named *oVirtSync* for handling the synchronization concern in oVirt-Engine and used *oVirtSync* to solve the problem of synchronization for some of the commands. We describe here in more detail the definition, implementation, and use of *oVirtSync*.

Language design Listing 1 shows the definition of *oVirtSync* expressed in the Xtext grammar definition format. The language model consists of `Command` elements. Each `Command` element includes:

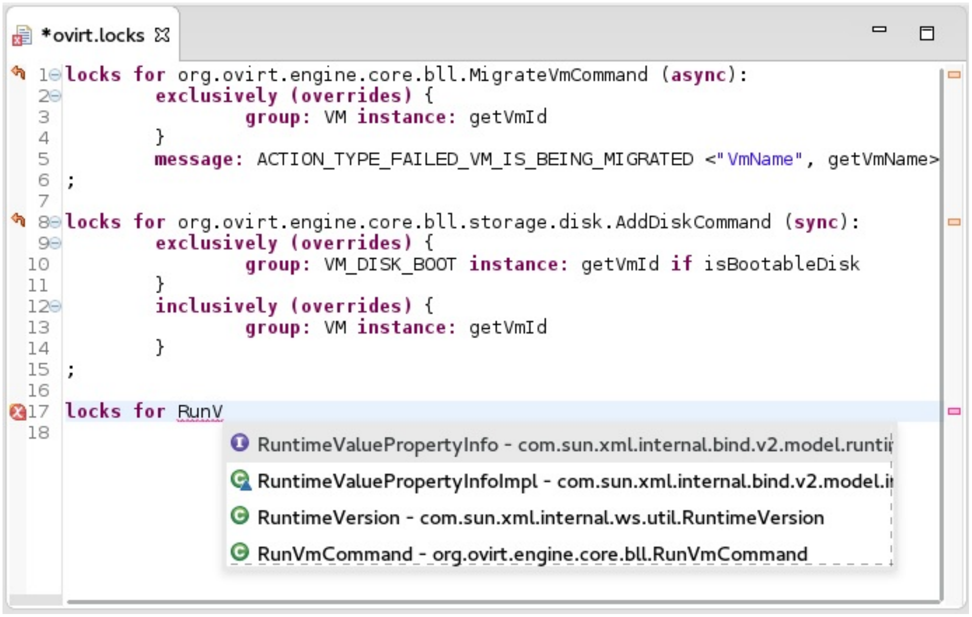
- The type of the command the locks are defined for.
- The scope of the locks.
- Whether or not the command is supposed to wait until all the locks can be acquired. If any of the locks cannot be acquired the command fails.
- Optionally, a list of read-locks.
- Optionally, a list of write-locks.
- Optionally, a message to display if another command cannot be executed due to the acquired locks.

Each `Lock` element inside the lock lists includes:

- The type of the entity to lock.
- The identifier of the entity instance to lock.

Listing 2: LocksGenerator.xtend

```
class LocksGenerator implements IGenerator {

 def compile(Resource resource) {
  this.resource = resource
 '''
  package org.ovirt.engine.core.bll;

  import java.util.*;
  import lom.runtime.BridgedSourceLocation;
  import org.ovirt.engine.core.common.action.LockProperties;
  import org.ovirt.engine.core.common.action.LockProperties.
       Scope;
  import org.ovirt.engine.core.common.locks.LockingGroup;

  public privileged aspect Locks {
  ... skipped ...
   «FOR cmd:resource.allContents.filter(typeof(Command)).
        toIterable»
    «cmd.compile»
   «ENDFOR»
  }
 '''
 }

 def compile(Command cmd) '''
  «NodeModelUtils.getNode(cmd).toSourcePosition»
  LockProperties around(LockProperties lockProperties, «
       cmd.type.qualifiedName» command): execution(*
       applyLockProperties(..)) && args(lockProperties)
       && target(command) {
   return lockProperties«cmd.scope.compile»«cmd.isWait.
        compile»;
  }

  «IF NodeModelUtils.getNode(cmd.exclusiveLocks) != null»
  «NodeModelUtils.getNode(cmd).toSourcePosition»
  Map<String, Pair<String, String>> around(«cmd.type.
       qualifiedName» command): execution(*
       getExclusiveLocks()) && target(command) {
   Map<String, Pair<String, String>> locks = new HashMap
        <String, Pair<String, String>>();
   «FOR lock:cmd.exclusiveLocks.locks»
    «lock.compile»
   «ENDFOR»
   return locks;
  }
  «ENDIF»

  «IF NodeModelUtils.getNode(cmd.sharedLocks) != null»
  «NodeModelUtils.getNode(cmd).toSourcePosition»
  Map<String, Pair<String, String>> around(«cmd.type.
       qualifiedName» command): execution(* getSharedLocks
       ()) && target(command) {
   Map<String, Pair<String, String>> locks = new HashMap
        <String, Pair<String, String>>();
   «FOR lock:cmd.sharedLocks.locks»
    «lock.compile»
   «ENDFOR»
   return locks;
  }
  «ENDIF»
 '''
  ... skipped ...
}
```



```
1  locks for org.ovirt.engine.core.bll.MigrateVmCommand (async):
2          exclusively (overrides) {
3                  group: VM instance: getVmId
4          }
5          message: ACTION_TYPE_FAILED_VM_IS_BEING_MIGRATED <"VmName", getVmName>
6  ;
7
8  locks for org.ovirt.engine.core.bll.storage.disk.AddDiskCommand (sync):
9          exclusively (overrides) {
10                 group: VM_DISK_BOOT instance: getVmId if isBootableDisk
11         }
12         inclusively (overrides) {
13                 group: VM instance: getVmId
14         }
15 ;
16
17 locks for RunV
18
             RuntimeValuePropertyInfo - com.sun.xml.internal.bind.v2.model.runtir
             RuntimeValuePropertyInfoImpl - com.sun.xml.internal.bind.v2.model.ir
             RuntimeVersion - com.sun.xml.internal.ws.util.RuntimeVersion
             RunVmCommand - org.ovirt.engine.core.bll.RunVmCommand
```

Figure 1: Programming with *oVirtSync*

- Optionally, a method that defines whether the lock should be acquired for the specific instance of the command or not.

The effort needed in order to implement the grammar definition and code transformation for *oVirtSync* was relatively low. We made no attempt to generalize it in order to be reusable for similar problems or other projects. Overall, the *oVirtSync* DSAL was relatively easy to design, since it only needed to resolve the synchronization problem as it manifests in oVirt-Engine.

Language implementation Listing 2 depicts a portion of the code transformation of *oVirtSync* to the kernel language that was developed in the Xtend programming language [1], a domain specific language for code transformation provided in Xtext. While implementing the grammar definition and code transformation we leveraged not only formats but also tools and IDE support that are provided by Xtext that significantly reduced the implementation effort. Furthermore, by transforming *oVirtSync* to the kernel language we avoided the complicated job of developing a weaver plugin that implements the weaving semantics needed for *oVirtSync*.

Language use With *oVirtSync* one should be able to define the required locks for all commands in oVirt-Engine. Figure 1 shows a screenshot of an aspect in *oVirtSync* as it appears in Eclipse. Lines 1-15 define the locks for two commands in oVirt-Engine, namely `MigrateVmCommand` and `AddDiskCommand`. `MigrateVmCommand` (lines 1-6) locks exclusively the VM with the identifier that it is returned by the `getVmId` method. The lock is released at the end of the entire command execution (the scope is set to "`async`" in line 1). The command fails if the locks cannot be acquired ("`wait`" is omitted in line 1). The message defined in line 5 is displayed if another command cannot be executed because of the acquired lock. Similarly, lines 8-15 define the locks for `AddDiskCommnad`. Note the use of a condition in line 10 to indicate that the exclusive lock should be acquired only if the disk is bootable.

During programming with *oVirtSync* we enjoyed full IDE support. This is thanks to an Eclipse plugin generated by Xtext for programming in *oVirtSync*. Auto-completion and syntax-highlighting can be seen in line 17. In addition, cross-reference markers (lines 1 and 8) are produced by AJDT. Other than tool support, the simple and declarative syntax of the language makes it easier to express the desired behavior.

5. Related Work

Elsewhere [9–11] we proposed a DSAL workbench as a tool that facilitates the development of first-class DSALs, i.e., DSALs that are as easy to develop as DSLs and are as effective to program in as GPALs. The DSAL workbench comprised a traditional language workbench as its front-end and an extended version of the AWESOME [14] composition framework (that supports multiple DSALs) as its back-end. The main difference between that approach and the approach presented in this paper is the fact that with a DSAL workbench one would still need to implement a weaver plugin whereas the approach here eliminates that need.

Some aspect frameworks transform DSALs into another language. In Reflex [20] DSALs are transformed into a kernel language. In XAspects [19] DSALs are transformed into AspectJ. However, transformation in general does not preserve the aspect behavior when multiple DSALs are being composed. To overcome this problem, the kernel language that we use provides constructs for resolving foreign advising and co-advising conflicts. Another difference between the kernel language in Reflex and the one we use is that ours is based on AspectJ and therefore AspectJ is supported out-of-the-box. Since the transformation preserves the original source location of advice and join point shadows defined within the DSAL code, development tools that work with AspectJ will also work with our DSAL code.

Interpreter-based frameworks like JAMI [12], POPART [2], and Pluggable AOP [13] also avoid low-level implementation of the weaving semantics. However, they achieve simplicity at the expense of performance, since their conflict resolution is based on interpretation. In our approach, the use of DSALs does not imply performance degradation compared to use of GPALs.

SPECTACKLE [18] is a tool that facilitates the resolution of multi-DSAL co-advising conflicts. There is a similarity between SPECTACKLE and our approach in the sense that both strive for a specification based DSAL composition process.

6. Conclusions

In this paper we bring the DSAL development process one step closer to the development process of DSLs, for a class of DSALs that are in a sense reducible to a GPAL. By doing so the effort required for defining, implementing, and using these DSALs is reduced to a level that makes LOM practical with respect to this class of DSALs.

From a language implementation perspective, the implementation cost of a DSAL is greatly reduced by replacing the development of a weaver (plugin) per DSAL with a transformation of the DSAL into a kernel language. Making use of a language workbench facilitates both the implementation of such a transformation and the definition of the DSAL grammar.

From a language definition perspective, the reduced cost of DSAL implementation relaxes the expectation that the produced DSAL be reusable. Since the effort required for developing a DSAL is reduced, this effort might be justified even when the DSAL cannot be reused in other projects. Therefore, it becomes cost-effective for one to design a DSAL that is specific to the problem at hand. The less reusable the DSAL is, the easier it is to design the DSAL (and to implement it).

From a language use perspective, programming in the DSAL becomes more effective. First, with the IDE plugin that is produced by the language workbench one gets editing tools like those available when programming with a GPAL. Second, when transforming the DSAL into a kernel language that is an extension of an existing GPAL, one is able to use GPAL development tools while programming in the DSAL.

References

[1] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[2] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 51–62, Charlottesville, Virginia, USA, Mar. 2009. ACM Press.

[3] A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with AJDT. In *Proceedings of the ECOOP 2003 Workshop on Analysis of Aspect-Oriented Software*, Darmstadt, Germany, July 2003.

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

[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, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[7] M. Fowler. Language workbenches: The killer-app for domain specific languages, 2005.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[9] 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.

[10] 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.

[11] 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.

[12] W. Havinga, L. Bergmans, and M. Akşit. Prototyping and composing aspect languages: using an aspect interpreter framework. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, number 5142 in Lecture Notes in Computer Science, pages 180–206, Paphos, Cyprus, July 2008. Springer Verlag.

[13] S. Kojarski and D. H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 247–263. ACM Press, 2005.

[14] S. Kojarski and D. H. Lorenz. Awesome: An aspect co-weaving system for composing multiple aspect-oriented extensions. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'07)*, pages 515–534, Montreal, Canada, Oct. 2007. ACM Press.

[15] S. Kojarski and D. H. Lorenz. Identifying feature interaction in aspect-oriented frameworks. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 147–157, Minneapolis, MN, May 2007. IEEE Computer Society.

[16] D. H. Lorenz. Language-oriented modularity through Awesome DSALs: summary of invited talk. In *Proceedings of the 7th AOSD Workshop on Domain-Specific Aspects Languages (DSAL'12)*, pages 1–2, Potsdam, Germany, Mar. 2012. ACM Press.

[17] 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.

[18] D. H. Lorenz and O. Mishali. SpecTackle: Toward a specification-based DSAL composition process. In *Proceedings of the 7th AOSD Workshop on Domain-Specific Aspects Languages (DSAL'12)*, Potsdam, Germany, Mar. 2012. ACM Press.

[19] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain specific aspect languages. In *Companion to the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 28–37, Anaheim, California, 2003. ACM Press.

[20] É. Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, number 4089 in Lecture Notes in Computer Science, pages 98–113, Vienna, Austria, Mar. 2006. Springer Verlag.

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