Untangling Crosscutting Concerns in Domain-specific Languages with Domain-specific Join Points

Tom Dinkelaker, Martin Monperrus, Mira Mezini

Untangling Crosscutting Concerns in Domain-specific Languages with Domain-specific Join Points

Tom Dinkelaker Martin Monperrus Mira Mezini
Technische Universität Darmstadt
{dinkelaker,monperrus,mezini}@informatik.tu-darmstadt.de

ABSTRACT

Like programs written in general-purpose languages, programs written in DSLs may also suffer from tangling and scattering in the presence of domain-specific crosscutting concerns. This paper presents an architecture that supports aspect-oriented features for domain-specific base languages. Both base programs and advices are written in different domain-specific languages. The framework relies on the concept of *domain-specific join point*.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Language Constructs and Features—*Frameworks*

General Terms

Design, Languages

Keywords

Aspect-oriented Programming, Domain-specific Languages

1. INTRODUCTION

In most aspect-oriented languages the concern-specific code is written in the same language as the base program, i.e., the part of the program that is not considered concern-specific. Although, crosscutting concerns may be more directly expressed using domain-specific abstractions. To address this problem, *domain-specific aspect languages* (DSALs for short) provide abstractions to implement aspects for a special domain in a declarative way. Several DSALs have been proposed, each targeting a particular domain (e.g., [16] for distributed software). These languages improve AOP technology by combining it with DSL technology.

However, the combination of AOP and DSL technology can be effective in the other direction as well, having aspects improve the modularity of DSLs. Like programs written in general-purpose languages, programs written in DSLs may also suffer from tangling and scattering in the presence of domain-specific crosscutting concerns. Several approaches provide aspects for DSLs in a particular domain (e.g., [18] for weaving in grammar specifications).

Most existing AO language implementations strongly depend on a particular *base language*: For each new base language, e.g., a new DSL, an AOP tool must be re-implemented specifically for this DSL, resulting in wasted development costs. Extensible aspect compilers [4, 3] and run-times [17, 22, 21] support reusing aspect implementation infrastructure for general purpose base languages such as Java or Scheme. Yet, none of them targets DSLs as the base language. On the other hand, none of the extensible DSAL approaches [19, 5, 12] supports composing a domain-specific advice language with a domain-specific base language.

In this paper, our contribution is an approach for defining aspect languages for DSLs. We present a framework, called POPART, that supports aspect-oriented features for domain-specific base languages. Both base programs and advices are written in different DSLs. The framework relies on the concept of *domain-specific join point*, that has been identified as an open question during the Third Workshop on Domain-Specific Aspect Languages [7].

The paper is organized as follows: Sec. 2 elaborates the need for domain-specific join point models. Sec. 3 elaborates how domain-specific aspect languages can be implemented for DSLs. Sec. 4 discusses related work and Sec. 5 concludes the paper.

2. PROBLEM STATEMENT

Domain-specific languages (DSLs) provide special abstractions that are closer to their problem domain, hence enable developers to express their intents more directly.

Consider a simple *workflow language* [8] for assembling composite applications from services inspired from BPEL [1]. The DSL, called *ProcessDSL*, provides abstractions to define processes that consist of several tasks to be executed in a sequence. ProcessDSL introduces several domain-specific keywords: the literal `registry` abstracts over the details of accessing the registry component; the operation `notify` is used to send an email message to all stakeholders of a process; `process` and `task` are domain-specific abstractions. Furthermore, Process DSL defines keywords for security enforcement: the two operations `encrypt` and `decrypt` are used to secure messages; literals (e.g., `RSA`) enable to select an appropriate encryption algorithm; and the literal `me` represents the identity on whose behalf the process is executed.

DSAL'09, March 3, 2009, Charlottesville, Virginia, USA.

```
1  process(name:"EasyCreditProcess") {
2    def offers = [:];
3    task (name:"getOffers") {
4      def services = registry.find("Banking");
5      services.each { bank ->
6        def enc_req = encrypt(new RateReq(),RSA,bank.pubKey);
7        def enc_resp = bank.call("getRate",[enc_req]);
8        offers [bank] = decrypt(enc_resp,RSA,me.privKey); ;
9    } }
10   task (name:"selectOffer") {
11     def bank = ... //get cheapest bank from offers
12     def enc_req = encrypt(new BorrowReq(),RSA,bank.pubKey);
13     def enc_resp = bank.call("borrow",[enc_req]);
14     def resp = decrypt(enc_resp,RSA,me.privKey);
15     notify "Borrowed a credit from ${bank}: ${resp}";
16 } }
```

Figure 1: An Example DSL for Workflows

Fig. 1 shows a *ProcessDSL* program that defines a process to select the cheapest from a set of credit offers. A process with name "EasyCreditProcess" is defined with the keyword `process` (lines 1–16). It consists of two tasks to be executed in the defined order. Each task in ProcessDSL has a name (defined in the round brackets following the keyword `task`) and a closure (the code block in the curly brackets following the name declaration). The code inside a task closure may contain DSL abstractions, e.g., `registry` in line 4, as well as general purpose code like `each{}` for iterating over collections (cf. line 5). The first task (lines 3 to 9) searches a list of banking services using the special keyword `registry`, invokes `getRate` on each of them, and stores the offered rates. The second task (lines 10–16) selects the cheapest rate (line 11) and invokes `borrow` on the selected banking service. It uses `notify` (line 15) to send an email with the credit details to all stakeholders of the process (line 16).

Charfi [6] showed that there are crosscutting concerns in workflow languages such as BPEL similar to the one presented here. The example shows that even for toy DSLs like ProcessDSL, programs written may easily suffer from tangling and scattering. In lines 6–8 and lines 12–14, functional code of the process is scattered and tangled with non-functional code of a security concern for confidential Web service communication. For the sake of separation of concerns, we would like to modularize the above crosscutting concern by using aspects for ProcessDSL programs.

However, aspect-oriented programming tools are generally available only for general-purpose languages. Similar to general-purpose languages, a solution for separation of crosscutting concerns in a DSL program is to separate the base program from the crosscutting concerns. However, in contrast to general purpose languages, the aspect weaver or composer has to take into account domain-specific join points.

Domain-specific join points are roughly points in the execution of a DSL program. In contrast to join points of general-purpose languages, they are rather related to actions that are performed on abstractions of a domain (e.g., the executing a domain operation on a domain object). In the context of ProcessDSL, the domain-specific events are considered in this paper are (but not limited to) the following: executions of a process or a task (similar to a method execution in AspectJ [2]), retrieval of services from the registry, and Web service calls (similar to a method call).

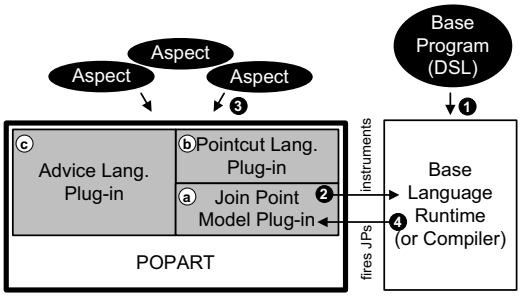

Figure 2: The DSAL Plug-in Architecture of popart.

3. ASPECTS FOR DOMAIN-SPECIFIC BASE LANGUAGES

The *Pluggable and OPen Aspect Run-Time* (POPART for short) is an open framework for defining aspect languages. POPART can be used to define aspect languages for any domain, as long as the base language infrastructure can be instrumented (cf Sec. 3.4). The support for aspects for domain-specific base languages is implemented as a set of plug-ins in POPART. A plug-in consists of several classes that define the syntax and semantics of a particular part of the aspect language. To implement a DSAL in POPART, the DSAL's implementer provides three plug-ins for: (a) the domain-specific join point model; (b) the domain-specific pointcut language; (c) the domain-specific advice language.

Fig. 2 shows this plug-in architecture for implementing aspect-oriented DSLs. The aforementioned plug-ins are shown as gray boxes (indices a–c). This architecture enables to reuse default aspect-oriented semantics provided by the framework. The addition of necessary domain abstractions are made on top of these default aspect-oriented mechanisms.

Embedded DSLs [14] are used to implement the domain-specific pointcut language and domain-specific advice language plug-ins. The point of using embedded DSLs is simplicity: the domain syntax is simply encoded as expressions of the host language and the domain-specific semantics are implemented as a library in the host language. The embedded DSL principle allows us: 1) to reuse the default aspect-oriented keywords and semantics for the DSAL, such as `aspect` and `around`; 2) general-purpose constructs in advice, as the reuse of `each` in Fig. 1; 3) general-purpose operators in pointcuts.

3.1 Domain-specific Join Point Models

Implementing an aspect language for a DSL requires to define what elements in a DSL semantics can be considered as possible join points. A join point is a point in the execution of a program, where an aspect can contribute functionalities.

Definition: A *domain-specific join point* is a join-point related to a domain abstraction (domain event, domain keyword, etc.).

Definition: A *domain-specific join point model* is a specification of all possible domain-specific join-points. A *domain-specific join point model* is used to bind the domain-specific aspect language to a domain-specific base language. A *domain-specific join point model* determines which points in the execution of DSL programs are visible to aspects and where aspects can contribute functionality.

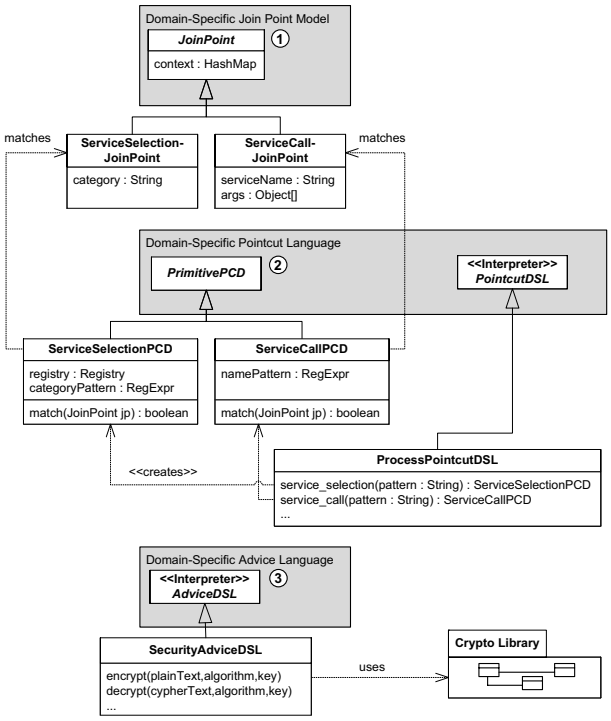

Figure 3: ProcessDSL's Join Point Model (excerpt).

A domain-specific join point model can be defined according to the following steps: (1) analyze the domain-specific base language to find relevant points in the execution of DSL base programs; (2) for each of these points, add a new kind of join point into the join point model; (3) for each kind of join point provide an *instrumentation module* that fires the join point to the aspect engine.

In POPART, a domain-specific join point model is defined by extending an object-oriented framework that models the generic aspect meta-model shared by all domain-specific plug-ins. Fig. 3 schematically shows the extension for implementing a specific join point model for the ProcessDSL language presented in Sec. 2. The gray boxes contain framework classes. The other classes correspond to the specification of a domain-specific join point model for ProcessDSL.

The ProcessDSL join point model consists of the following domain-specific join points, each with a specific context:

process execution Points at which a process instance is executed. The context exposed to advices contains: the process name, the task list.

task execution Points at which a task of a process is executed. Context is: the task name, the activities.

service selection Points at which the registry is consulted to select services of a certain category. Context is: the category pattern that is used to select services and the resulting set of selected services.

service call Points at which a service is called, e.g., a remote call to a Web service via its *service proxy*. Context is: the name of the service being called, the arguments of the call, whether the invocation is remote or local, and the result of the invocation.

Let us now consider the example of service call join points. For implementing this join point type, we add the class `ServiceCallJoinPoint` of Fig. 3 to the join point model of ProcessDSL. Next, we have to provide an adequate instrumentation module. Specifically, the ProcessDSL interpreter contains a class `ServiceProxy` that is responsible for invoking Web service, i.e., when encountering a service call operation in DSL code, the `ServiceProxy.call(...)` will be called, which generates a SOAP message that is send out a Web service. At this point, the instrumentation must fire instances of class `ServiceCallJoinPoint` to the POPART engine. We use an AspectJ [2] aspect to instrument the DSL interpreter classes. E.g., for service call, we have implemented an aspect that extracts context information out of the interpreter classes, creates a `ServiceCallJoinPoint`, and fires it to POPART. For the other join points, similar instrumentation modules fire instances of either `ServiceSelectionJoinPoint`, `ProcessExecutionJoinPoint`, or `TaskExecutionJoinPoint`

3.2 Domain-Specific Pointcut Languages

For writing pointcuts for a corresponding join point model within POPART, a *pointcut language* plug-in has to be implemented. The domain-specific pointcut language enables to select join points from a domain-specific join point model. A domain-specific pointcut language reduces the semantic gap that exists when using a DSAL composed of general-purpose pointcut language and domain-specific advice language.

Definition: A *domain-specific pointcut language* is a language that enables the specification of pointcuts using high-level, domain-specific keywords. It can be attached to a *domain-specific join point model*.

A domain-specific pointcut language is implemented as a subclass of `PointcutDSL` of the framework, as shown Fig. 3. While the domain independent operation on pointcuts can be reused from `PointcutDSL`, e.g. boolean operations, new keywords are defined for new domain-specific pointcut designators. For instance, for ProcessDSL, a subclass `ProcessPointcutDSL` is created that extends `PointcutDSL` and implements keyword methods, such as `service_call()` and `service_selection()`.

In the following, a pointcut expression is given in the pointcut language for the workflow DSL. The pointcut expression matches all remote invocations to services whose operation name matches the regular expression with "get.*":

service_call("get.∗") & **if_pcd** { **external** }

The pointcut expression composes `service_call` and `if_pcd` using "&". While the pointcut designator `service_call()` selects all service calls that operation name match the regular expression "get.*", the `if_pcd()` pointcut designator checks whether the service call is a remote call.

The pointcut language plug-in implements the keyword semantics in `ProcessPointcutDSL`. For each pointcut designator keyword, a method is implemented with the corresponding keyword name, which takes the pointcut parameters, and returns an instance of `PointcutDesignator` that is used to match against join point instances. E.g., on receiving `service_call(...)`, the pointcut language interpreter creates a `ServiceCallPCD` object (Fig. 3), passing the regular expression "`get(.)*`". The `ServiceCallPCD` selects all `ServiceCallJoinPoint`s with service names matching its regular expression. Note that the semantics of the general-

```
aspect(name:"SecurityAspect") {
  around (service_call(".*") & if_pcd { external }) {
    encrypt(request,RSA,thisJoinPoint.service.pubKey);
    def enc_resp = proceed();
    response = decrypt(enc_resp,RSA,
      thisJoinPoint.process.privKey);
} }
```

Figure 4: A Security Aspect for ProcessDSL.

purpose designator is reused from the superclass `PointcutDSL`, that defines the corresponding keyword methods. Note that in `if_pcd` the join point context can be accessed from the condition expression inside an if-closure. In the above pointcut expression, because `ServiceCallJoinPoint` defines the property `external`, this join point context variable `external` is accessible in the `if_pcd` subexpression.

Note that, implementing a `PointcutDSL` as plug-in, which is referenced by the advice language interpreter, has an important advantage: One can reuse both the pointcut and advice languages independently from each other. Further, a pointcut is simply an instance of a `Pointcut` subclass and can be composed into a hierarchical structure using the pointcut combinatorsNote also that since POPART uses embedded DSLs, i.e., pointcuts are simply expressions in the host language, it is possible to extend the pointcut language with new keywords without affecting other parts of the aspect language implementation.

3.3 Domain-Specific Advice Languages

Definition: A *domain-specific advice language* is a language that enables to write advice with domain-specific keywords and abstractions.

In order to create a fully-fledged domain-specific aspect-oriented runtime environment, a third plug-in implementing an *domain-specific advice language* has to be implemented. This plug-in is integrated with a domain-specific pointcut language and a (domain-specific) join point model.

In POPART, domain-specific advice languages are implemented as subclasses of `AdviceDSL`. For security concerns, we define the `SecurityAdviceDSL` class that extends the framework at extension point 3 in Fig. 3. Most important, the domain-specific advice language interpreter implements DSL keywords as methods that have the corresponding keyword name (e.g., `encrypt()`). Moreover, the domain-specific contexts discussed above are made available for advice implementations. E.g., the variables `request` and `response` implicitly refers to the outgoing and incoming message.

Fig. 4 shows the crosscutting concern identified in Sec. 2 as a domain-specific aspect. Generic keywords of the POPART framework are used, e.g. `around`[1]. The pointcut is written in the domain-specific pointcut language presented above. The advice is written in the domain-specific advice language `SecurityAdviceDSL`.

3.4 Implementation Details

POPART extensively uses embedded DSLs [14] implemented in Groovy [11]. We chose Groovy as an implementation language because it supports various advanced language features, such as, flexible syntax, closures, and a meta-object protocol, that enable the rapid implementation [9] of embedded DSLs. Embedded DSLs are used for domain-specific pointcut languages and domain-specific advice languages.

The boot process of POPART is as follows: first, the POPART runtime is loaded; then, DSL programs are loaded (Fig. 2, index 1) and instrumented using the instrumentation modules – currently AspectJ aspects (index 2); next, domain-specific aspects are loaded (index 3); finally, during the execution of DSL programs, the instrumentation will fire join points (index 4) to the aspect kernel that composes in the aspects.

At runtime, everything is seamlessly integrated in the same virtual machine: DSL programs, domain-specific aspects and the POPART kernel. In a nutshell, the join points that are fired are matched against the defined pointcuts and if there is a match the corresponding advice is executed. Note that the details of aspect composition are out of scope of this paper, we refer to [10]. Domain-specific aspects may contain keywords of several DSLs. When POPART executes the domain-specific aspects, it dispatches the encountered keywords to the corresponding plug-in that defines the semantics. Aspect keywords are dispatched to: 1) the plug-in instance of `AdviceDSL`, 2) the plug-in instance of `PointcutDSL`, 3) the POPART kernel, 4) the base DSL infrastructure.

The ProcessDSL discussed in this paper has been implemented as an embedded DSL, but this is not required. The base DSL infrastructure only has to be available as Java bytecode (e.g., implemented in Java or Groovy), this is because AspectJ is used to instrument the DSL infrastructure.

4. RELATED WORK

The *abc* compiler [4] allows to define new kinds of pointcuts as extensions to the AspectJ semantics, but is not targeted for DSALs and only weaves on Java.

The *Aspect SandBox* (ASB) [17] is a framework for prototyping alternative AOP semantics implemented in Scheme. ASB has been improved with an extensible join point model [22]. ASB allows defining new kinds of join points at the cost of heavy impacts in the weaver implementation.

Several DSAL frameworks have been proposed for general-purpose base languages, such as Java: *Reflex* [21], AweSome [15], JAMI [12], and Dinkelaker et. al. [9]. They have certain support for the composition of different extensions and resolving conflict between them, however the differences in their support for implementing DSALs are not important for a comparison. They all only employ a join point model for Java. Because new kinds of join points are out of scope, weaving on DSLs is not supported.

XAspects [19] is an extensible system that defines a DSL as an aspect. Other aspects are plug-ins that extend DSL implementations with particular concerns. XAspects uses special pointcuts for traversals, but cannot define new kinds of pointcuts. XAspects uses the join point model of AspectJ and does not support new join point types.

Strembeck and Zdun defined [20] an aspect-oriented DSL for role-based access control. While they make an ad-hoc implementation using a dynamic aspect language, our contribution is generic. POPART can be used to *aspectize* any DSL. Moreover, its plug-in architecture allows us to use different languages for base programs and advice.

To our knowledge, only Heidenreich et al. proposed [13] a generic approach for aspect-oriented DSLs that is based on invasive software composition, i.e., source code rewriting. Our paper differs on two points: 1) the POPART framework

[1] we use well-established aspect-oriented *pointcut-and-advice* semantics for DSLs that are similar to AspectJ [2] semantics.

is dynamic, hence, allows to load and unload domain-specific aspects at runtime, 2) while Heidenreich et al. assume that the base program and the advices are written in the same language, POPART allows to use different domain-specific languages in the base program and the advice.

5. CONCLUSION AND FUTURE WORK

We presented the POPART framework that supports the implementation of aspect-oriented DSLs. It is based on the concept of domain-specific join point model. This concept enables to deal with aspect composition only based on domain relevant events. POPART provides a plug-in architecture for implementing aspect languages for domain-specific languages.

A DSL can be integrated with the framework by defining a domain-specific join point model, a domain-specific pointcut language, and a domain-specific advice language. POPART relieves the DSAL designer of implementing common aspect semantics – only additional domain abstractions and semantics must be integrated into the framework.

We have implemented a domain-specific aspect language for a simple workflow language as an application case. The sample DSL's design has been inspired from complex workflow languages, such as BPEL. The aspect-oriented programming prototype for the DSL is implemented (whereby reusing existing libraries for complex domain operations such as encryption) in three plug-in consisting of 15 classes, with less than 500 LOC. The prototype supports common aspect-oriented features, such as general-purpose pointcut designators (e.g., `and`, `or`, `not`, `cflow`, `if`) and advice keywords (e.g, `thisJoinPoint`, `proceed`). Moreover, the ProcessDSL prototype comes with other features of POPART discussed in [10]: semantic variation in aspect languages at runtime; dynamic aspects; and resolution of aspect interactions.

Future work will address the current limitations: 1) measuring and improving execution speed 2) implementing missing features such as inter-type declarations, and 3) composition of different domain-specific advice languages and the resolution of possible conflicts between language definitions.

The POPART source code and the implementation of the example ProcessDSL can be downloaded from: `http://www.stg.tu-darmstadt.de/popart`.

6. ACKNOWLEDGMENTS

This work was partly supported by the *feasiPLe* project, Federal Ministry of Education and Research, Germany.

7. REFERENCES

[1] A. Arkin, S. Askary, B. Bloch, et al. Web Services Business Process Execution Language 2.0, OASIS Standard, 11 April 2007.

[2] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[3] P. Avgustinov, T. Ekman, and J. Tibble. Modularity First: A Case for Mixing AOP and Attribute Grammars. In *AOSD'2008*, 2008.

[4] P. Avgustinov, J. Tibble, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, and G. Sittampalam. Abc: An Extensible AspectJ Compiler. In *AOSD'2005*, 2005.

[5] A. Bagge and K. Kalleberg. DSAL = Library+Notation: Program Transformation for Domain-Specific Aspect Languages. In *DSAL Workshop*, 2006.

[6] A. Charfi. *Aspect-Oriented Workflow Management: Concepts, Languages, Applications*. VDM Verlag Dr. Mueller, 2008.

[7] T. Cleenewerck, J. Noyé, J. Fabry, A.-F. Lemeur, and E. Tanter, editors. *Summary of the Third Workshop on Domain-Specific Aspect Languages (DSAL'08)*, 2008.

[8] T. Dinkelaker, A. Johnstone, Y. Karabulut, and I. Nassi. Secure Scripting Based Composite Application Development: Framework, Architecture, and Implementation. In *Conference on Collaborative Computing*, 2007.

[9] T. Dinkelaker and M. Mezini. Dynamically Linked Domain-Specific Extensions for Advice Languages. In *DSAL'2008*, 2008.

[10] T. Dinkelaker, M. Mezini, and C. Bockisch. The Art of the Meta-Aspect Protocol. In *AOSD'2009*, 2009.

[11] The Groovy Home Page. `http://groovy.codehaus.org/`.

[12] W. Havinga, L. Bergmans, and M. Aksit. Prototyping and Composing Aspect Languages Using an Aspect Interpreter Framework. In *Proceedings of ECOOP'2008*, page 180. Springer, 2008.

[13] F. Heidenreich, J. Johannes, and S. Zschaler. Aspect Orientation for Your Language of Choice. In *Proc. Workshop on Aspect-Oriented Modelling at MoDELS 2007*, 2007.

[14] P. Hudak. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28(4es):196–196, 1996.

[15] S. Kojarski and D. Lorenz. AweSome: An Aspect Co-Weaving System for Composing Multiple Aspect-Oriented Extensions. In *OOPSLA'2007*, 2007.

[16] C. Lopes. *D: A Language Framework For Distributed Programming*. PhD thesis, Northeastern University, 1997.

[17] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In *Proc. CC 2003*, volume 2622 of *LNCS*, 2003.

[18] D. Rebernak, M. Mernik, H. Wu, and J. Gray. Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammar. In *DSAL'06*, 2006.

[19] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An Extensible System for Domain Specific Aspect Languages. In *OOPSLA*, 2003.

[20] M. Stremberck and U. Zdun. Definition of an Aspect-Oriented DSL using a Dynamic Programming Language. In *Proceedings of the Workshop Open and Dynamic Aspect Languages'2006*, 2006.

[21] E. Tanter and J. Noyé. A Versatile Kernel for Multi-language AOP. *GPCE 2005*, 2005.

[22] N. Ubayashi, H. Masuhara, and T. Tamai. An AOP Implementation Framework for Extending Join Point Models. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2004.