A Taxonomy of Domain-Specific Aspect Languages

JOHAN FABRY, PLEIAD Lab, Computer Science Department (DCC), University of Chile
TOM DINKELAKER, Ericsson R&D
JACQUES NOYÉ, INRIA - Ecole des Mines de Nantes - LINA
ÉRIC TANTER, PLEIAD Lab, Computer Science Department (DCC), University of Chile

40

Domain-Specific Aspect Languages (DSALs) are Domain-Specific Languages (DSLs) designed to express crosscutting concerns. Compared to DSLs, their aspectual nature greatly amplifies the language design space. We structure this space in order to shed light on and compare the different domain-specific approaches to deal with crosscutting concerns. We report on a corpus of 36 DSALs covering the space, discuss a set of design considerations, and provide a taxonomy of DSAL implementation approaches. This work serves as a frame of reference to DSAL and DSL researchers, enabling further advances in the field, and to developers as a guide for DSAL implementations.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Languages

Additional Key Words and Phrases: Aspect-oriented programming, domain-specific languages

ACM Reference Format:
Johan Fabry, Tom Dinkelaker, Jacques Noyé, and Éric Tanter. 2015. A taxonomy of domain-specific aspect languages. ACM Comput. Surv. 47, 3, Article 40 (February 2015), 44 pages.
DOI: http://dx.doi.org/10.1145/2685028

1. INTRODUCTION

In large software systems, some concerns cannot be properly encapsulated into their respective modules. The implementation of these so-called *crosscutting concerns* cuts across the module structure of the application. As a result, the benefits of modular software development are largely compromised. Aspect-Oriented Programming (AOP) is a paradigm to modularize such crosscutting concerns [Kiczales et al. 1997]. An *aspect* is a new kind of module that captures a crosscutting concern. The ability of an AOP language to support crosscutting lies in three elements, defined as follows (verbatim from Masuhara et al. [2003]):

(1) The *join points* are the points of reference that programs including aspects can affect. *Lexical* join points are locations in the program text (e.g., "the body of a method"). *Dynamic* join points are runtime actions, such as events that take place during execution of the program (e.g., "an invocation of a method").

This work was partially funded by the Inria Associate Team RAPIDS/REAL.
Authors' addresses: J. Fabry and É. Tanter, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Santiago, Chile; emails: {jfabry, etanter}@dcc.uchile.cl; T. Dinkelaker, Ericsson R&D, Frankfurt, Germany; email: tom.dinkelaker@gmail.com; J. Noyé, ASCOLA Team, INRIA - Ecole des Mines de Nantes - LINA, Nantes, France; email: Jacques.Noye@mines-nantes.fr.

DOI: http://dx.doi.org/10.1145/2685028

(2) A means of *identifying* join points (e.g., "the bodies of methods in a particular class," or "all invocations of a particular method").
(3) A means of *effecting* at join points (e.g., "run this code beforehand").

Lexical join points are also sometimes called static join points [Rebernak et al. 2009], contrasting the words static and dynamic. To clarify the difference between static (lexical) and dynamic join points in the preceding examples: the static (lexical) join point would have information of, for example, the filename and the line numbers of the method, while the dynamic join point would have information on the actual values of the parameters, receiver object, and so on. For uniformity, in the remainder of this text we shall exclusively use the terminology static join point.

The preceding three elements altogether form what is known as the *join point model* of an AOP language [Masuhara et al. 2003], although this term is not used consistently in the literature (e.g., Wand et al. [2004] differ). To implement their crosscutting support, AOP languages provide an *aspect weaver*, which processes the program such that join points are actually created, identified, and effected upon when needed.

The design space of aspect languages can be described in terms of join points, their identification, and effects at join points. For instance, in AspectJ [Kiczales et al. 2001], join points are dynamic and correspond to the different actions during program execution, such as method executions, object creations, and so forth. Join points are identified by means of *pointcuts*, which are declaratively defined predicates on join points. Finally, the means of effecting at join points is the *advice* mechanism: An advice associates a pointcut with a body construct. Whenever a join point of interest occurs, the advice body is executed, similarly to implicit invocation mechanisms. Additionally, an advice *kind* specifies whether the body is executed before, after, or instead of the join points.

While many aspect languages closely resemble AspectJ in the use of its pointcut-advice model to address crosscutting, some differ. For instance, the means of identifying join points need not be expressed in a declarative sublanguage. Instead, join points can be identified through pointcuts defined in a general-purpose programming language, for example, as in AspectScheme [Dutchyn et al. 2006], or through annotations added (manually) on program elements. The means of effecting at join points may also be purely declarative, by opposition to the imperative nature of AspectJ advice bodies. For example, in the Program Description Logic (PDL) language [Morgan et al. 2007], the effect specification is a piece of plain text (as will be discussed in Section 2.2).

In addition, each of the three previous elements (join points, identification, effect) may be tailored to a specific domain. Actually, historically the first aspect languages were domain specific [Lopes 1997; Mendhekar et al. 1997]. Domain-Specific Languages (DSLs) [van Deursen et al. 2000; Mernik et al. 2005; Fowler 2010; Ghosh 2010; Mernik 2013] offer dedicated abstractions relevant to the domain at hand, with many benefits for comprehension [Kosar et al. 2010, 2012], maintenance [van Deursen and Klint 1998], verification, and optimization [Bruce 1997; Basu 1997]. Many Domain-Specific Aspect Languages (DSALs) have been proposed in the literature but it is often difficult to compare and relate them precisely. This is because their domain specificity makes it hard to see the underlying mechanisms involved to support crosscutting. Nevertheless, DSALs have to include the three elements common to all aspect languages: join points, join point identification, and means of effecting at join points. As it turns out, each of these elements may or may not be domain specific, resulting in many combinations of DSAL designs.

The goal of this article is to structure the DSAL design space in order to shed light on and compare the different domain-specific approaches to deal with crosscutting concerns. The two extra dimensions of domain specificity—join points and their

identification—are new in the language design space. Hence, the main focus of this text is on the three axes of join points, join point identification, and means of effecting at join points. Other elements will be touched upon, but a comprehensive exploration of all possible elements in the design space is out of the scope of this text.

To perform this study, we have selected and examined a corpus of 36 DSALs published in conferences and workshops directly related to the community working on Aspect-Oriented Software Development (AOSD). In order to stay focused and study a well-defined corpus, we consider neither modeling languages nor languages published outside the community that may also be considered to address crosscutting issues. We have also surveyed the implementation approaches for DSALs as reported in the literature, including infrastructures specifically created for the definition of DSALs. Example languages presented in the work of implementation approaches are *not* included in the 36 DSALs of our study. This is because they are typically minimalistic and presented only as an illustration of the features of the approach.

We intend this work to be a general frame of reference for research on DSALs and an enabler of further advances in the field. We provide the following contributions:

(1) We propose definitions for and give a synopsis of properties of DSALs. We define the terms domain-specific aspect language, domain-specific join points, domain-specific join point representations, and domain-specific pointcut language. We define a three-dimensional space relating the domain-specific nature of join points, join point identification, and means of effecting at join points. Lastly, we summarize published motivations for the use of domain-specific languages and show how they apply to DSALs.
(2) We give an overview of existing DSALs based on the preceding definitions. Not all DSALs are detailed in this article. Instead, we restrict ourselves to five representative languages summarized in the beginning of the text that set the scene and provide key points for study. We do, however, provide an online companion in the form of an annotated bibliography of all these languages, which is available at http://dsal.cl/inventory.
(3) We discuss a set of considerations that should be taken into account when designing a DSAL and which choices have been taken by existing languages.
(4) We provide an implementation taxonomy that is based on existing taxonomies for DSLs. The taxonomy is split in five branches. For each branch, we give a general description, provide an example, and discuss the trade-offs. Furthermore, we discuss a completely new implementation mechanism: infrastructures for DSALs and give an overview of six such infrastructures.

This article is intended to serve as a guide to DSAL and DSL researchers and developers. It may also be of interest to a much broader community as a toolbox of languages providing solutions to specific needs as well as a possible way to address specific crosscutting concerns for which no DSAL has yet been developed. DSAL researchers can benefit from the frame of reference, which allows them to more clearly situate their work in the design space as well as to identify areas in the space that have been lacking in attention. DSAL developers can use the elaborated design decisions and the implementation classification as a guide for their effort. For DSL researchers and developers, this article also provides food for thought as well as material for practical developments. Indeed, this work provides a new perspective on DSLs that are not standalone languages but rather extensions to a base language.

This article is organized as follows: In the following section, we illustrate the advantages of using DSALs as well as the breadth of the area by summarizing five example languages. Section 3 proposes a succinct definition of the term DSAL and discusses key properties before giving an overview of all languages we studied. In Section 4, we

provide a guide to the design of DSALs by outlining the main DSAL design decisions. Then, in Section 5, we to implementation of DSALs, classifying the different implementation techniques into a DSAL implementation taxonomy, including recommendations of use. The article then concludes.

2. A HANDFUL OF DSALS

In this section, we briefly present five example DSALs: RG, PDL, KALA, AspectG, and AspectMatlab. These examples first serve to illustrate how the use of DSALs aid programmers by allowing them to succinctly express the specification of a possibly complex crosscutting concern thanks to the use of domain-specific abstractions. Second, the different examples illustrate the breadth of the area, as each of these languages has distinctive features. Third, these languages provide key data points for the analysis of the design space and serve as reference examples in the DSAL design discussion that will follow.

2.1. DSAL Example: RG

Reverse Graphics (RG) [Kiczales et al. 1997; Mendhekar et al. 1997] is historic by virtue of being the first aspect language discussed in the seminal "Aspect-Oriented Programming" paper by Kiczales et al. Indeed, DSALs lie at the origin of AOP. While the intent of the papers that talk about RG are clearly focused toward arguing for the need of AOP, the RG language itself is described in sufficient detail to allow us to give an overview of the language here.

2.1.1. Domain and Goals. RG was designed in the context of an image processing system. An RG application is built by composing predefined image filters built from primitive filters. These primitive filters rely on a small number of loop structures iterating on the pixels of the input images and performing simple operations (e.g., logical operations on these pixels). As a result, an application can be seen as a dataflow graph whose nodes correspond to the primitive filters and whose edges correspond to the connections between filter outputs and filter inputs. With each node consuming and producing images, a direct implementation of such a graph results in "excessively frequent memory references and storage allocation, which in turn leads to cache misses, page faults and terrible performance" [Kiczales et al. 1997].

The goal of RG is to optimize memory usage and eliminate redundant computations through the use of three aspects: First is memorization, which avoids multiple applications of the same filter to the same images. Second is fusion, which performs loop fusion when primitive filters using identical loop structures are composed. Third is memory management, which tries to reuse the memory allocated to temporary results [Mendhekar et al. 1997].

2.1.2. Example. The following code illustrates what happens at the heart of the fusion aspect:

```
(cond ((and (eq (loop-shape node) 'pointwise)
            (eq (loop-shape input) 'pointwise))
       (fuse loop input 'pointwise
               :inputs (splice ...)
               :loop-vars  (splice ...)
               :body  (subst ...))))
```

The first two lines of the code test whether the loop shapes of the filter node and of the input are both pointwise. If this is the case, the remainder of the code fuses the loops, that is, a new loop is generated with the same structure where inputs (line 4), loop variables (line 5), and bodies (line 6) are merged.

2.1.3. Anatomy of the Language

Join Points. The join points used by RG are domain specific: They are the invocations of the primitive filters. The join points are represented as the dataflow graph mentioned previously so that the aspects have access to the necessary contextual information, for instance, to the loop structure of the filter arguments.

Join Point Identification. Join points in RG are identified using Lisp, that is, a general purpose language. While the code written in this language treats elements of the domain, it has the full expressiveness of Lisp at its disposal. Hence, the join points themselves are domain specific, while the means to denote them, that is, the means to perform join point identification, is general purpose.

Effecting at Join Points. Effects in RG are written in a domain-specific language defined as "a simple procedural language that provides simple operations on nodes in the dataflow graph" [Kiczales et al. 1997].

2.1.4. Implementation. RG is implemented in Lisp. Weaving is done by a preprocessor that combines base code and aspects to produce C code. The weaver "uses unfolding as a technique for generating a data flow graph from the component program" [Kiczales et al. 1997]. The aspects manipulate this graph directly and the result is passed to a C code generator.

2.1.5. Remarkable Properties. The discussion of RG mentions the difference in code size between a "manually tangled version" of the code and the AOP code. The former consists of 35,213 lines of code, while the AOP version consists of 1,111 lines of code. If we also add the implementation of the weaver, 3,520 lines, it still yields over seven and a half times less code. This illustrates the conciseness advantage that can be gained using a DSAL. Moreover, this conciseness does not come at a significant cost: The paper states that the performance of both the manually tangled and AOP version are comparable. Lastly, the AOP version is over 100 times faster than the naive implementation that does not optimize computation.

2.2. DSAL Example: PDL

The second DSAL example we present is a further illustration of how DSALs allow the implementation of a concern using extremely concise code. In PDL [Morgan et al. 2007], specification of effects at join points solely consists of a string, as we show next.

2.2.1. Domain and Goals. Design rules for programs are used to formalize desired programming practices. These rules define constraints on the structure or the behavior of the program, ranging from stylistic guidelines up to how libraries should be used. Programs can be statically checked for violations of these rules using purpose-built tools called *design rule checkers*. The stated goal of PDL is to allow programmers to easily specify design rules as pointcut-advice pairs. Pointcuts express when a design rule is violated; advice states the corresponding error message. These rules can then be used to verify .Net assemblies (bytecode libraries): For each matched pointcut, the corresponding error message is printed.

2.2.2. Example. The reference paper contains various examples, where the error message usually suffices to establish the purpose of the example. Consequently, we simply include four of these examples here.

```
sourceType && public && nested
  : "Nested types should not be visible"

constructor(sourceType && abstract) && public
  : "Abstract types should not have public constructors"
```

```
method(sourceType) && 'void *.Finalize()' && !protected
  : "Finalizers should be protected"

sourceNamespace && < 5 type
  : "Avoid namespaces with few types"
```

2.2.3. Anatomy of the Language

Join Points. In PDL, join points are the static elements of the program under verification: classes, methods, namespaces, and so forth. As these are written in a general-purpose language, we can consider that the join points correspond to the static join points from general-purpose aspect languages.

Join Point Identification. PDL defines a pointcut language that reuses elements of the AspectJ pointcut language syntax and adds elements required for the domain of program rule checking: existence, universality, and also cardinality checking. As a result of these extensions we can state that the pointcut language is a domain-specific language.

Effecting at Join Points. Specifying the effects in PDL solely consists of the error message that is printed when pointcuts match, that is, when a design rule is violated. We, therefore, consider that effects are specified in a domain-specific manner.

2.2.4. Implementation. The PDL rule checker is a preprocessor that transforms the code of the rules to C# code. This is then compiled and run on the .Net assembly, performing bytecode analysis to find violations of the design rules.

2.2.5. Remarkable Properties. PDL is an example of a DSAL where the code for effects at join points is reduced to an absolute minimum, solely specifying the error message that needs to be shown. This illustrates how a DSAL can allow for extremely concise code and that code specified for effecting at join points may be purely declarative.

2.3. DSAL Example: KALA

In the previous two examples, we have shown languages that at least have general-purpose join points or general-purpose join point identification. The next example, KALA [Fabry et al. 2008], shows that the domain specificity of a DSAL can exhibit itself by having join points, identification, and effects all be domain specific. It also shows that the definition of effects can express significantly complex domain-specific entities or computations.

2.3.1. Domain and Goals. Advanced transaction models address the shortcomings of the classical model of database transactions, due to the original design goal to provide concurrency management for short and unstructured data accesses. There are many such advanced models, each specifically tailored to address one shortcoming. Chrysanthis and Ramamritham [1991] defined a formalism, called *ACTA* that allows for a formal specification of how a given model modifies or adds to these properties. KALA takes this model and allows programmers to attach extra transactional properties to parts of the application that are meant to run within a transaction. As a result, the program uses the advanced transaction model that is defined in the KALA program.

2.3.2. Example. Arguably the best-known advanced transaction model is nested transactions. It enables a running transaction T to have a number of child transactions T_c. Each T_c can view the data used by T. When a T_c commits the data is delegated to the parent; when it aborts its changes are ignored. Data is only committed to the

database if the root of the transaction tree commits. This is specified in the following code, mapping the formal specification to a given set of Java methods.

```
util.strategy.Hierarchical.child*() {
  alias(parent Thread.currentThread() );
  name(self Thread.currentThread());
  groupAdd(self "ChOf"+parent);
  begin { dep(self wd parent); dep(parent cd self); view(self parent);}
  commit { del(self parent);
           name(parent Thread.currentThread()); terminate("ChOf"+self); }
  abort { name(parent Thread.currentThread()); terminate("ChOf"+self);} }
```

The first line of the preceding code is the signature, which identifies the transactional methods. The remainder is the body, which we do not discuss in detail here, as it would take us too far into transactions specificities. We just note the presence of the three phases in the lifecycle of a transaction: begin, commit, abort, and the use of the ACTA formal concepts of dependencies (dep), views (view), and delegation (del) in the body.

2.3.3. Anatomy of the Language

Join Points. Join points are domain specific: transaction executions, and within their scope the reads and writes of persistent values.

Join Point Identification. KALA defines a domain-specific language to specify pointcuts. It identifies transactions by specifying the signature of a method whose execution corresponds to a transaction execution. This automatically includes reads and writes of persistent values in the scope of the transaction.

Effecting at Join Points. Effects are specified in a domain-specific language: an executable variant of the ACTA formal model for advanced transactions.

2.3.4. Implementation. Two implementations of KALA have been realized. The first weaver was an ad hoc preprocessor that transformed Java and KALA source code to Java source code. A second implementation, called ReLAx [Fabry et al. 2009], uses the DSAL infrastructure capabilities of the Reflex AOP kernel [Tanter and Noyé 2005].

2.3.5. Remarkable Properties. KALA illustrates the extreme domain-specific form that a DSAL can take. Since the effects are specified in a derivation of a formal model for advanced transaction management, programmers need to have knowledge of this model to use KALA. For such domain experts, it supports the succinct and modular definition of new transactional models.

2.4. DSAL Example: AspectG

A domain for which a significant number of DSALs have been proposed is that of (domain-specific) programming language implementation [Klint et al. 2005; Kalleberg and Visser 2006; Rebernak et al. 2009]. It should perhaps not be surprising that language implementation researchers implement their own DSALs. We show one such example here, AspectG [Rebernak et al. 2009]. It shows that crosscutting concerns do not only present themselves in general-purpose languages but also in DSLs, and hence that a DSAL can consider a DSL as base programming language.

2.4.1. Domain and Goals. In the domain of programming languages, definition and compiler construction special tools and languages are used to provide the language specifications. These typically allow a compiler to be generated based on the specification of the grammar productions of the language, which also specify semantic actions for each grammar production. In such specifications, modularity is typically based on the

language syntax, that is, grammar productions. However, some language features can crosscut these productions, for example, type checking [Rebernak et al. 2009].

AspectG is a DSAL for ANTLR [Parr and Quong 1995], a language grammar and tool intended to facilitate the construction of development tools for a given language (e.g., debuggers). AspectG allows for the definition of additional tools on top of an existing ANTLR language specification by adding the additional semantic actions required for these tools inside the grammar productions. These tools may be written in any programming language and it is frequent for them to add similar semantic actions across (parts of) the grammar [Rebernak et al. 2009].

2.4.2. Example. The example given in the paper presents a toy DSL for the control of a robot. It defines a number of grammar productions, where those grouped in the command namespace represent the commands that can be given to the robot. Some semantic actions of these productions contain the string fileio.print(”time=time+1;”) to log the time taken by the robot when executing that command. We do not include the complete definition here for space reasons (and as a result the presented excerpt is not well formed). The following example adds a mapping from the line numbers in the DSL to the line numbers in the code generated by the tool, which is useful, for example, for debuggers.

```
pointcut count_gpllinenumber():
  within(command.*) && match(fileio.print("time=time+1;"));

after():  count_gpllinenumber() {
  gplendline=fileio.getLinenumber();
  filemap.print("mapping.add(newMap(" + dsllinenumber + ",Robot.java,"
               + gplbeginline + "," + gplendline + "));"); }
```

The example shows that AspectG closely follows the pointcut-advice model. The pointcut `count_gpllinenumber` matches all productions in the command namespace whose semantic actions contain the string fileio.print(”time=time+1;”). The advice adds two Java statements at the end of the block of semantic actions of the matched rule. These are responsible for maintaining the mapping from the DSL source code to the generated program code.

2.4.3. Anatomy of the Language

Join Points. The base language for AspectG is ANTLR: a DSL used to specify the grammar productions and the semantic actions. The join points are both the grammar productions and the semantic actions, that is, these are domain specific.

Join Point Identification. AspectG uses a domain-specific pointcut language as well: Pointcuts are (composition of) filters on grammar productions or semantic rules.

Effecting at Join Points. Advice is written in a general-purpose language. Its contents, defining the semantic actions, is written in Java or, possibly, any (typically general-purpose) language.

2.4.4. Implementation. Appropriate to a DSAL for programming language grammars, the implementation of the language is done through transformation of programming language grammars. Aspects are woven into the ANTLR grammar itself, in a preprocessor approach. The preprocessor uses an ANTLR parser together with low-level matching and transformation operations on the abstract syntax tree of the ANTLR source.

2.4.5. Remarkable Properties. Beyond having a DSL as the programming language of the base code, AspectG is remarkable in that it is not inherently restricted to one

language to specify the effects. The authors state that the effect language can be any general-purpose language as long as it is the language used to define the semantic actions of the parse tree. This flexibility comes from the fact that effecting at join points in AspectG solely consists of adding the semantic actions defined in the advice to the relevant parts of the parse tree.

2.5. DSAL Example: AspectMatlab

The last example DSAL we present here, AspectMatlab [Aslam et al. 2010], shows how the domain specificity of a DSAL can be more subtle than what was presented in the previous examples.

2.5.1. Domain and Goals. Matlab is a programming language targeted at numerical computing. It provides features such as high-level array operations and contains specific optimizations for matrix operations (e.g., when using sparse matrices). We consider Matlab to be a DSL because of its target audience, the numeric features, and especially their optimization. We, however, acknowledge that this choice may be controversial and discuss this in more detail in Section 3. AspectMatlab aims to add aspects to Matlab, both for the typical general-purpose language constructs as well as for the matrix and loop operations typical of numerical computing. "One of the main goals of AspectMatlab is to expose these language constructs to aspect-oriented programming in order to make it appropriate for use in the scientific domain." [Aslam et al. 2010].

2.5.2. Example. To be more aligned to Matlab syntax, AspectMatlab calls the specification of join point identification *patterns* and join point effects *actions*. Patterns are enclosed within a patterns end block and actions are enclosed within an actions end block. A simple example from the paper is as follows:

```
patterns
  pCallFoo : call(foo) & within(function, bar);
end
actions
  aCountCall : before pCallFoo
    %action code
  end
end
```

The pattern shows the use of the within keyword that restricts matching lexically to the type and name of entities given as parameter. In this case, the pattern matches calls to a function foo from within a function called bar.

The AspectMatlab paper provides three example aspects: tracking the sparsity of arrays, measuring floating point operations, and adding the treatment of SI units[1] to computations. We do not include such an example here because of their relatively large size and complexity.

2.5.3. Anatomy of the Language

Join Points. The join point types are function call and execution, array read and write operations, as well as execution of loops, loop headers, and bodies. All these are general-purpose language constructs.

Join Point Identification. The language for patterns includes selectors for each kind of join points (e.g., loophead denotes the execution of a loop header). Consequently, join point identification is performed in a general-purpose way.

[1]Metre, kilogram, second, Ampere, Kelvin, candela, and mole.

Effecting at Join Points. Actions in AspectMatlab are written in Matlab. As Matlab is a DSL, effects at join points are specified in a domain-specific manner.

2.5.4. Implementation. The AspectMatlab weaver is a source to source preprocessor, the output of which is plain Matlab code. Matlab is dynamically typed, with the syntax for function call the same as for array access. This makes it impossible to statically establish the semantics of such an expression without performing control flow analysis. The weaver performs such an analysis, making it possible to resolve a significant amount of cases. Unresolved cases are dealt with runtime tests [Aslam et al. 2010].

2.5.5. Remarkable Properties. AspectMatlab is remarkable as a DSAL that could arguably be considered as a general-purpose aspect language. This is because the language for describing effects is Matlab and Matlab could be considered a general-purpose rather than domain-specific language. This illustrates that there is a source of ambiguity in the DSAL classification, which we further discuss in Section 3. Also notable here is that the effect language is the same as the base language, showing that a DSAL may not need to define a new language or constructs to express means of effecting at join points.

3. A DEFINITION AND OVERVIEW OF DOMAIN-SPECIFIC ASPECT LANGUAGES

Despite a significant amount of DSALs having been developed, we are not aware of any accepted definition of what it means for a language to be a DSAL. To lay the groundwork for discussion and clarify basic principles for a wide audience, we elaborate such a definition and discuss basic properties in this section.

In the survey paper of Mernik et al. on DSLs [2005], DSLs are defined as follows:

> DSLs are languages tailored to a specific application domain.

The survey paper furthermore succinctly states the advantages of DSLs over General-Purpose Languages (GPLs) as follows: "By providing notations and constructs tailored toward a particular application domain, they offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs. Also, by reducing the amount of domain and programming expertise needed, DSLs open up their application domain to a larger group of software developers" [Mernik et al. 2005].

The survey does not give, in turn, a definition of an *application domain*. For the sake of completeness, we quote the definition given by Rolling in his annotated bibliography on domain engineering [1994]:

> A *domain* is the term used to refer to a set of related functions and problems with a common vocabulary, terminology and a common set of strategies for describing requirements and for providing solutions to those problems. The set of applications common to the different functions may be properly considered the scope of the domain.

More recently, Heering and Mernik have defined two notions of application domain: *type A domains* and *type B domains*. Type A domains are "a field of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in the field" and type B domains are "a system family, that is, a set of software systems that exhibit similar functionality." [Heering and Mernik 2007].

DSALs differ from DSLs in that they are aspect languages, that is, that they aim to modularize a crosscutting concern. The notion of *concern* and *separation of concerns* refers to the idea that, when designing or implementing a system, the work should be split in units relating to the "natural concerns" of the application domain rather than

in units imposed by technology (the *solution domain*). This idea can be traced back to Parnas [1972] and Dijkstra [1982]. The concept of *crosscutting concern* originally surfaced as an implementation issue: the inability of standard decomposition techniques, for example, in functions or objects, to properly modularize some concerns [Kiczales et al. 1997; Tarr et al. 1999]. Crosscutting concerns have since been shown to also surface in requirements analysis and design, which is also known as "early aspects." Work on early aspects is, however, out of the scope of this article; our focus is on implementation, and part of the analysis that can be properly supported by a programming language and its environment.

Taking the preceding information into account, we state the following concise definition for DSALs:

> A DSAL is a DSL that features language mechanisms for expressing crosscutting concerns.

Concern-specific languages are related to DSALs. They are defined by Bodden as follows: "A concern-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to or in support of, a particular crosscutting concern, comprising implicit or explicit quantification over events in the dynamic control flow" [Bodden 2005]. While this definition in essence only differs slightly from the preceding DSAL definition, the terminology "concern-specific language" did not catch on. We only found one use of it in the literature [Braem et al. 2007], where the concept apparently was elaborated independently. Consequently, we find the term DSAL and the earlier definition to be more appropriate and we consider concern-specific languages as being DSALs.

We now consider in more detail what new elements DSALs bring to the DSL design space before discussing the advantages of creating a new language versus building abstractions in an existing language. We end this section with an overview of the languages we studied.

3.1. Domain-Specific Pointcut Languages and Join-Point Representation

The definition of a DSAL highlights the distinctive element of DSALs: crosscutting concerns. Key to a DSAL is allowing some form of quantification over the code that is being crosscut, that is, the base code. To summarize the AOSD terminology, we introduced in Section 1, the structure and computation of the base code is reified as *join points*, and the aspects consider these to produce their effects. The DSAL allows for the specification of *join point identification* that selects a subset of these where the structure or behavior of the DSAL program, the *effect at join points*, is placed or executed, respectively. As the DSAL is a DSL, the obvious assumption to make here is that the language in which the effect is specified is a DSL. A DSAL can, however, also be domain specific with regard to the join points being considered and how these are identified. For example, in Section 2, we saw that KALA and AspectG define domain-specific join points and join point identification, and PDL defines a domain-specific join point identification.

The aforementioned is a new and distinctive element of DSALs: join points and a means of identifying them. These create two extra dimensions in the domain specificity of the language. First, the join points emitted by the base code can reflect domain-specific abstractions instead of general-purpose ones, such as method invocations or variable accesses. Second, join points can be identified using a notation tailored to an application domain.

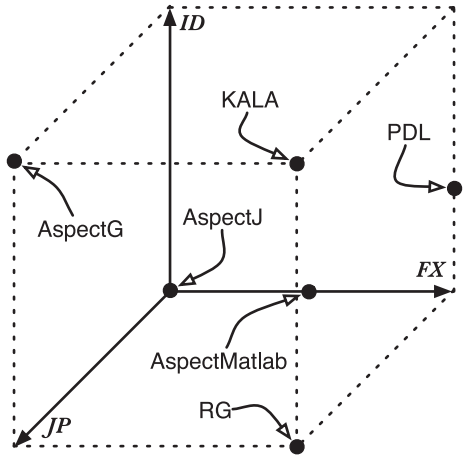

Fig. 1. The three axes of a DSAL: Join Points (JP), Means of Join Point Identification (ID), and Means of Effecting at Join Points (FX), with the origin signifying general-purpose and the domain-specific nature increasing along the axes. The languages of the previous section and AspectJ are positioned in the space.

Note that the DSAL definition we propose does not specify which of these three dimensions is domain specific. As we have shown in Section 2, different combinations are possible and previous work has intended to establish the independence of these three dimensions [Cleenewerck et al. 2008]. In Figure 1, we illustrate the three-dimensional space created by the axes. Furthermore, it positions the languages we discussed in Section 2 at their approximate position in this space, together with AspectJ [Kiczales et al. 2001], the prototypical general-purpose aspect language (GPAL). Note that we do not claim that the three axes are continuous, thus allowing for domain specificity of two languages to be compared using their position on the axes. At most, there are different degrees of domain-specific nature; even establishing a partial order between different languages is far from trivial.

The two extra dimensions of domain specificity (join points and their identification) are new in the language design space. Hence, they require a new kind of design decision to be made in the language creation process. To the best of our knowledge, these decisions and their impact have not yet been studied, and this article provides a first step in this direction by structuring the DSAL space. To do this, we require a clear concept of what it means for join points and their identification to be domain specific, which is explained next.

3.1.1. Domain-Specific Join Points. If a join point is the materialization of a concept of a specific domain instead of a general-purpose concept, that is, a different join point kind, then we consider it a domain-specific join point. Hence the definition:

A domain-specific join point is a join point whose kind is a domain-specific concept.

An example of such a domain-specific join point can be found in RG (discussed in Section 2.1). Join points in RG correspond to a manipulation of an image, which are implemented by filter operations on the pixels of the image.

It is informative to consider how domain-specific join points can be introduced. For this, let us consider the join point generation process which occurs when aspects are woven. Recall that join points can be static or dynamic. In both cases, one can consider a *join point stream* to refer to the sequence of join points that is created during weaving. For static join points, this is when processing the program text, and for dynamic join

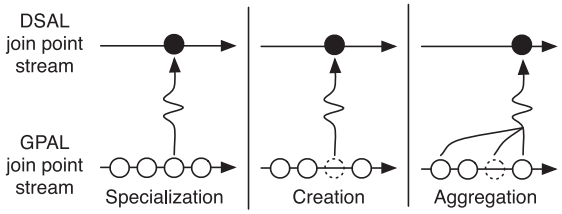

Fig. 2. Three types of domain-specific join point generation: abstraction of existing join points, creation of new join points, and aggregation of join points.

points this is when executing the program. When these join points are general purpose (either because they are produced by a GPL or because they correspond to nonspecific join points produced by a DSL), we identify three ways to generate domain-specific join points: specialization, creation, and aggregation (Figure 2):

—Specialization consists in specializing a general-purpose join point to the join point stream of the DSAL. For example, in KALA [Fabry et al. 2008], the general-purpose join point of a *method execution* is represented as a domain-specific abstraction: a *transaction boundary*. In the specialization process context information may be added or removed (e.g., removing the list of arguments of a method called join point).
—Creation consists in generating a join point for an event that is otherwise not present as a join point in the stream of the base language and adding it to the stream of the DSAL. An example of this is the "internal join points" in AO4BPEL [Charfi and Mezini 2007]: The base language only describes the combination of activities, without considering the actions of the activities themselves. Internal join points expose some of these actions as join points to the AO4BPEL language, that is, create new join points at the DSAL level.
—Aggregation consists in aggregating a number of general-purpose join points, possibly taking into account events that are not reified as join points in the GPAL. The aggregate is then added to the stream of join points of the DSAL. Note that this is similar to aspect language proposals that take the history of join points into account, such as stateful aspects [Douence et al. 2002]. The difference here lies in the materialization of a domain-specific concept, absent in these languages. An example of aggregation is present in KALA where a transactional read or write join point is a combination of a transactional method execution join point (identifying a transaction) and a getter or setter method execution in its control flow (identifying the data read or written).

3.1.2. Defining Domain-Specific Join Point Representations. For clarity of the discussion, we wish to emphasize the distinction between a given (set of) join point(s) and all the join points that are considered by a DSAL. To achieve this, we rely on the terminology *join point representation*, which we define as follows:

> The join point representation of an aspect language determines the kind of points in the lexical structure or in the execution of the program where the aspects may have an impact.

The term join point representation was initially used in the original "Aspect-Oriented Programming" paper [Kiczales et al. 1997]: "Aspect weavers work by generating a join point representation of the component program and then executing (or compiling) the aspect programs with respect to it."

Importantly, we do not use the term *join point model* here to refer to the join point representation, as is sometimes the case (e.g., in Wand et al. [2004]). This is because

the term join point model is not used consistently in the literature. It is also used to refer to the trio join point representation, join point identification, and effecting at join points (e.g., in Masuhara et al. [2003]. We wish to avoid any confusion.

Based on our specification of the domain specificity of join points we can succinctly define domain-specific join point representation as follows:

> A domain-specific join point representation is a join point representation that includes at least one kind of domain-specific join point.

A corollary of the previous definition that is worth noting is the following: If a DSAL uses a representation that denotes a subset of the join points denoted by an existing general-purpose representation, we consider its representation as being general purpose. For example, in the RemoteJ language [Soule et al. 2007], the representation solely consists of method invocations and exception throwing and we consider it as general purpose.

3.1.3. Defining Domain-Specific Pointcut Languages. The large majority of the DSALs we studied perform join point identification and the specification of effects at join points in a similar fashion as in AspectJ. Put differently, the large majority of DSALs identify join points by means of pointcuts and use advice to specify the effect at join points. For languages in this framework of pointcut-advice model, we say that pointcut identification is performed by a pointcut language and the effects are defined through an advice language. Hence, it is useful to define what it means for a pointcut language to be domain specific.

A pointcut language can be considered as already being a domain-specific language tailored to the domain of "selecting join points.' Actually, pointcut languages of GPALs typically provide domain-specific abstractions (e.g., constructs for identifying specific kinds of join points). However, it is desirable to distinguish between pointcut languages for all types of base applications versus pointcut languages that focus on a specific domain of applications. Hence, we propose the following definition:

> A domain-specific pointcut language is a pointcut language that is tailored to a specific domain.

A wide variety of designs can be imagined for such a language. This ranges from adding domain-specific predicates to an existing pointcut language to building a complete domain-specific pointcut language. For example, in Section 2.2, we saw that PDL adds cardinality checks to (a restricted version of) a general-purpose pointcut language, while the KALA pointcut language solely consists of the possibility to specify transaction boundaries.

3.2. Building Abstractions versus a DSAL

An alternative to building a DSL is defining a library of domain-specific abstractions in a GPL. The latter requires less effort but lacks several advantages of DSLs. We repeat the most salient advantages listed in the survey paper of Mernik et al. on DSLs [2005] here:

(1) The DSL can provide *domain-specific notations* such that the intent of the code is clear. This may not be feasible in the GPL (e.g., due to syntax restrictions). This is a key feature, as it is directly related to the productivity improvement that DSLs yield [Mernik et al. 2005]. In addition to Mernik et al., Fowler provides a similar observation, stating that "The heart of the appeal of a DSL is that it provides a means to more clearly communicate the intent of a part of a system." [2010]

(2) The domain-specific constructs may *not be abstractable into one GPL construct* or into a single piece of GPL code. Typical examples are error handling and graph traversals. While a GPL cannot succinctly express the domain-specific constructs, the DSL can.
(3) As the domain-specific concepts are available as such, the language implementation may perform *domain-specific analysis, verification, and optimization*. In a GPL this may be harder or even unfeasible, depending on the construction of the concepts and the optimization possibilities of the language.

Fowler [2010] has a slightly different point of view, giving a number of additional advantages, and we highlight two of them here:

(4) DSLs ease *communication with domain experts* by providing a clear and precise language to deal with the domain. The biggest gain lies in letting domain experts read the code to confirm if it specifies the expected behavior.
(5) DSLs can use an *alternative computational model*, deviating from the typical imperative paradigm to whatever is more appropriate for the domain. A classic example of this is a DSL for defining state machines.

A number of the aforementioned advantages of DSLs naturally carry over to DSALs, especially when considering the specification of join point effects. The novelty of DSALs, however, lies foremost in the two other axes of domain specificity: join points and a means of identifying them, which is the focus of this article. DSL advantages in regard to these axes are as follows:

Join Point Representation. This is not a specification performed by the DSAL programmer, it is a specification at the level of the language definition. Hence for the programmer it does not contain limits on its notation, nor does it need to be executable or even have an execution context. Considering (2), mapping to constructs in the GPL, this is exactly what is addressed by the creation of domain-specific join points: The GPAL does not produce any suitable event, hence the DSAL creates one. Furthermore, the generation of domain-specific join points may allow for irrelevant context items to be removed, optimizing the generation process (3), which is discussed in more detail in Section 4.6. Using domain-specific concepts to define the join points will help communication between programmers using the DSAL and the domain experts (4). Lastly a domain-specific model for join points and their generation can be used (5) (e.g., join points for transitions in a state machine).

Means of Identifying Join Points. The advantages of the domain specificity of the representation carry through to the identification means. It is possible to use domain-specific notations (1) to also select join points not present in the GPAL (2). Identification can be verified and optimized using domain information (3), is more readable for domain experts (4), and can follow a domain-specific paradigm (5) (e.g., matching on transitions of a state machine).

3.3. Overview of DSALs

Table I lists all the DSALs we studied for this survey together with their reference paper. In Table II, we detail the domains they address, the domain-specific nature of their join point representation, identification, and effects, and we identify the base language. As a brief analysis of Table II, we consider the domain-specific nature of the axes of join point representation, identification, and effects. There are 13 languages where all three are domain specific, 19 where two of them are domain specific, and four where one is domain specific. Restricting to the languages where the base language is

Table I. The List of DSALs Studied (in Alphabetical Order) and the Reference to the Most Relevant Corresponding Article

Languages are identified by the last part of the URL of their summary description, which almost always corresponds to the name of the language. Prefix it with http://dsal.cl/dsal/ to obtain the full URL.

Name	Reference	Name	Reference
Alert	Bagge et al. [2006]	DiSL	Marek et al. [2012]
ALPH	Munnelly and Clarke [2008]	DSAL_for_Matlab	Cardoso et al. [2010]
AML	Irwin et al. [1997]	ERTSAL	Sousan et al. [2007]
AO4BPEL	Charfi and Mezini [2007]	KALA	Fabry et al. [2008]
AO4FSM	Dinkelaker et al. [2012]	LARA	Cardoso et al. [2012]
AO4SQL	Dinkelaker [2011a]	Leasing_in_MANets	Gonzalez Boix et al. [2008]
AOWP-PHP	Hokamura et al. [2008]	Padus	Braem et al. [2006]
AspectASF	Klint et al. [2005]	PCSL	Bruntink et al. [2005]
AspectG	Rebernak et al. [2009]	PDL	Morgan et al. [2007]
AspectGrid	Sobral and Monteiro [2008]	POM	Caromel et al. [2008]
AspectLisa	Rebernak et al. [2009]	Quo-ASL	Duzan et al. [2004]
AspectMatlab	Aslam et al. [2010]	Racer	Bodden and Havelund [2008]
AspectStratego	Kalleberg and Visser [2006]	RemoteJ	Soule et al. [2007]
CommentWeaver	Horie and Chiba [2010]	RG	Kiczales et al. [1997]
Conspects	Holzer et al. [2011]	RIDL	Lopes [1997]
COOL	Lopes [1997]	Robust	Fradet and Südholt [1999]
D4OL	Timbermont et al. [2008]	ScatterML	White and Schmidt [2009]
DAJ	Sung and Lieberherr [2002]	VMADL	Haupt et al. [2009]

domain specific, in eight of them the nature of all three axes is domain specific, while in five languages two axes have the domain-specific nature, and one language has only one axis whose nature is domain specific.

An overview of the different implementation approaches for DSAL construction is given in Section 5, and in Section 5.6, Table III classifies the different languages according to their implementation. Note that we did not include example languages presented as part of the implementation papers in our study. These languages are typically not developed in depth and are rather meant as illustrations of the features of the implementation approach. We consider that, as a consequence, they do not contribute significant data on the design space of DSALs, and we omit them from our study.

4. DSAL DESIGN DECISIONS

How and when to design a DSL is a topic that has already been discussed in the literature (e.g., in the work of Mernik et al. [2005]). Because they are dedicated to handle crosscutting, DSALs add new dimensions to this design space. As such, not only the specification of effects at join points but also the means of identifying join points as well as the join point representation need to be defined.

In this section, we explore the language space that is distinctive for DSALs. We focus on the three axes of Figure 1: join points, their identification, and means of effecting at join points. We outline the main DSAL design decisions related to the three axes and analyze the different choices that have been explicitly discussed or implicitly presented in the literature.

We start with a note on symmetry or asymmetry for DSALs, as it is a core language decision where we made a remarkable observation. This is followed by a discussion of previously presented design decisions. We then consider more in detail the three axes and discuss their independence followed by the degree of domain specificness. This is succeeded by an observation of the possible absence of pointcuts. An exploration of optimizations of domain-specific join points follows, and we end this section with considerations on composing programs written in various DSALs.

Table II. Overview of DSALs Studied, Providing the URL of Their Summary Description and the Domain Addressed

Includes General-Purpose (GP) or Domain-Specific (DS) nature of REP, join point representation; ID, join point identification; FX, effects at join point; Base, the base language. The references for the languages marked with * are AmbientTalk [Dedecker et al. 2006], EventJava [Eugster and Jayaram 2009], Lisa [Mernik et al. 2002], ANTLR [Parr and Quong 1995], Stratego [Visser 2004], ASF + SDF [van den Brand et al. 2001], and BPEL [Andrews et al. 2003].

http://dsal.cl/dsal/	Domain	REP	ID	FX	Base
COOL	Distributed systems	GP	DS	DS	GP: Java
RIDL		GP	DS	DS	GP: Java
Quo-ASL		DS	DS	GP	GP: Any
RemoteJ		GP	DS	DS	GP: Java
Leasing_in_MANets		DS	GP	DS	DS: AmbientTalk*
Conspects		DS	DS	GP	GP: EventJava*
PCSL	The software development process itself	GP	DS	DS	GP: C
Alert		GP	DS	DS	GP: C
PDL		GP	DS	DS	GP: .Net
CommentWeaver		GP	DS	DS	GP: Java
Robust		GP	DS	DS	GP: C-like languages
AspectLisa	Program transformation, compiler construction	DS	DS	DS	DS: Lisa*
AspectG		DS	DS	GP	DS: ANTLR*
AspectStratego		DS	DS	DS	DS: Stratego*
AspectASF		DS	DS	DS	DS: ASF + SDF*
AML	Numerical and scientific computation	DS	DS	DS	DS: Matlab subset
AspectMatlab		GP	GP	DS	DS: Matlab
DSAL_for_Matlab		GP	DS	DS	DS: Matlab
POM	Concurrent and parallel computing	DS	DS	DS	GP: Java
AspectGrid		GP	DS	DS	GP: Java
KALA		DS	DS	DS	GP: Java
Racer	Analysis of software	DS	DS	GP	GP: Java
DiSL		DS	DS	DS	GP: Java
LARA	Embedded systems	DS	GP	DS	GP: C and others
ERTSAL		GP	GP	DS	GP: C++
AO4BPEL	Workflows in the BPEL language	DS	GP	DS	DS: BPEL*
Padus		DS	GP	DS	DS: BPEL*
D4OL	Virtual machines	DS	DS	DS	GP: C (unspecified)
VMADL		DS	GP	GP	GP: C and others
RG	Image transformation	DS	GP	DS	GP: Lisp-like
ScatterML	Software deployment	DS	DS	DS	DS: nodes
ALPH	Healthcare	DS	DS	DS	GP: Java
AOWP-PHP	Web applications	DS	DS	DS	DS: PHP
DAJ	Adaptive programming	GP	DS	GP	GP: Java
AO4SQL	The SQL language	DS	DS	DS	DS: SQL
AO4FSM	Finite state machines	DS	DS	DS	DS: FSM

4.1. Asymmetric or Symmetric DSALs

Most aspect-oriented programming languages follow an asymmetric model, in which "aspects" are woven into "components" of the base [Harrison et al. 2002]. Aspects differ from components in their structure (e.g., the former identify and effect at join points, the latter do not). Components are composed differently than aspects, and sometimes

aspects cannot be composed with aspects, that is, the execution of aspect code cannot be intercepted by other aspects. In contrast, in a symmetric paradigm, "all components are treated as first-class, coequal building-blocks of identical structure, and in which no component's object model is more basic than any others" [Harrison et al. 2002]. Remarkably, we found that existing DSALs are universally asymmetric. All of them treat aspects as a different kind of entity than base code. This universal adherence to asymmetry, a core language decision, is, however, nowhere argued for in the literature. We, therefore, elaborate on it here.

We presume this universal asymmetry is because of three reasons: (1) symmetrical languages require the base and aspect language to be codesigned, (2) the language to define effects is usually domain specific, and (3) asymmetric GPALs are predominant.

(1) DSALs are typically designed taking into account an existing base language (arguably a similar situation to the design of GPALs). In symmetrical AOP languages, the base and aspect language features are codesigned. Hence, the base language may need to be changed, which can be problematic.
(2) In the large majority of DSALs, the language to define effects is a DSL (as shown in Table II). Hence, the aspects are fundamentally different entities from the base entities, implying asymmetry.
(3) In cases where the language to define effects is the same language as the base language—for example, AspectMatlab (see Section 2.5)—a symmetric approach may be possible. However, we found that in such cases the aspects are still treated differently, that is, following an asymmetric paradigm. In the AspectMatlab example, aspect code does not produce any join points [Doherty 2010], making it impossible for aspects to match on computation of (possibly other) aspects. Thus there is a difference between base and aspect code, and they are not "co-equal building-blocks" per the definition of Harrison et al. [2002]. The predominance of the asymmetric paradigm in GPALs may cause this design decision to be implicitly made. Again, AspectMatlab is an example of this: A more symmetric approach was never even considered in the design [Doherty 2010].

In conclusion, we do not see any impediment to building DSALs that are symmetric (e.g., when the language to define effects is the same language as the base language). However, in practice, we find that all DSALs are asymmetric and there is no argumentation given for this. A DSAL designer should at least be aware of this design choice and explicitly make or document it.

4.2. Seven DSAL Design Decisions

In early work on DSALs, Rebernak et al. [2009] introduce AspectLisa and AspectG (for AspectG, see also Section 2.4). As part of this article, the authors present the first analysis of the design space of DSALs by proposing seven design decisions that need to be made when elaborating a DSAL. The focus of the work is not to explore the design space, yet it is a good initial overview of design decisions. Hence, we review it here and place the design decisions in the context of this survey article. Note that Rebernak et al. seemingly use the join point model definition of Masuhara [2003] of join points, identification, and effect.

(1) What are the join points that will be captured in the DSAL?
(2) Are the DSAL join points static or dynamic?
(3) What granularity is required for these join points?
(4) What is an appropriate pointcut language to describe these join points?
(5) What are advice in this domain?
(6) Is extension/refinement only about behavior, or also structure?

(7) How is information exchanged between join points and associated advice (context exchange)? Is parametrization of advice needed?

Rebernak et al. do not elaborate further on these choices, as it is not within the scope of their article. Our interpretation, based on their article as well as the literature we studied, is as follows:

(1) The kind of points in the lexical structure or in the execution of the program where the crosscutting concern has an impact need to be established. Additionally, join point aggregation may be required (see Section 3.1.1). Aggregated join points contain some information of different parts of (the execution of) the program. This may be either for selecting the correct join points, or providing extra information to the effects. Examples of languages that perform join point aggregation are AO4BPEL [Charfi and Mezini 2007], KALA [Fabry et al. 2008], and Racer [Bodden and Havelund 2008].

(2) DSLs may contain a significant structural part, that is, where a large part of the DSL program represents structures. A DSAL for such languages may, therefore, wish to capture parts of these definitions as join points. As noted before, Rebernak et al. call these static join points, and they are also known as lexical join points [Masuhara et al. 2003]. In contrast, dynamic join points represent events in the execution of the base program. An example of the use of static join points in DSALs is as follows: In ANTLR, languages are specified by their grammar, the grammar productions are the static join points. AspectG allows additional semantic actions to be specified for selected grammar productions (see Section 2.4).

(3) We consider the granularity as a consequence of the choice of join points as regions in time versus points in time, as discussed by Masuhara et al. [2006]. The former are AspectJ-like join points that represent the duration of an event (e.g., the extent of an execution of a method), while the latter represent one instant in time (e.g., the moment a method is about to be executed). If the join points are considered as a region in time, this implicitly determines their granularity to be the execution time of the join point. Similarly, if join points are considered as being a point in time, these implicitly have the lowest possible granularity, as they have no duration.

(4) As a means for identification, the pointcut language is the element that typically receives the most attention in the literature we studied. Usually the first three design decisions are only briefly mentioned, or even absent. For example, 15 of the 36 DSALs we studied lack a specification of the join point representation. An example of a brief mention is present in the languages discussed in the AspectG paper, for example, "join points in ANTLR are static points in the language specification where additional semantic rules can be attached.' This is followed by explaining the elements of the representation, that is, the join points, by means of example pointcuts that match them. Note that the choice of including such a discussion is independent of the domain of the language: In AspectMatlab [Aslam et al. 2010] (see also Section 2.5), the discussion is absent, but in DSAL for Matlab [Cardoso et al. 2010] it is present.

(5) To adhere to the terminology defined in Section 3.1.3, we rephrase this item as "What are the effects in this domain?". The article does not elaborate much on the meaning of this design decision, not going deeper than a generic consideration of the behavior that needs to be specified in the aspect. The effects that can be performed, together with the choice of an effect language should be made here. What needs to be taken into account is when these effects need to be executed for region-in-time models, typically before, after, or around a region (see (1) and (3), and how join point information is passed (see (7)).

(6) GPALs not only have the possibility to change the behavior of the application but also its structure (e.g., in AspectJ intertype declarations allow fields and methods to be added to given classes). This functionality allows for modular extension of existing

classes, such that state and behavior of the crosscutting concern is added to the different modules where it conceptually belongs. Alternatively, they may also be used to address typing issues (e.g., by declaring that a given class implements a specific interface). DSALs may need to be able to perform such structural extensions as well. Examples of such languages are Alert [Bagge et al. 2006] and DAJ [Sung and Lieberherr 2002].

(7) For the specified effects to be able to correctly take place, this may require context information of the join point (e.g., the arguments of a method call). This, therefore, needs to be taken into account, typically by use of parametrization of the effects.

It is notable that in the previous list of design decisions none of them are tied to the domain that the DSAL is addressing. It is indeed possible that two different DSALs for the same domain differ in some of these choices. One example of this is AspectLisa and AspectG [Rebernak et al. 2009]. These languages both use the pointcut-advice model and differ in 5 of the 7 points: The join points (1) and their granularity differ (3), while being static in both cases (2). The pointcut languages are different (4). The advice are considered to be of the same kind (5), yet AspectLisa only extends behavior, while AspectG creates new tools that extend both structure and behavior of the application (6). Lastly, AspectLisa advice is parametrized, and in AspectG it is not (7). Another remarkable example of languages in a narrow domain but with different decisions is AO4BPEL [Charfi and Mezini 2007] and Padus [Braem et al. 2006]. Both languages target modularization of BPEL processes, yet join points (1) and their granularity (3) differ and the pointcut languages (4) are fundamentally different as well.

Rebernak et al. also present a rationale for the differences and similarities in design decisions between AspectLisa and AspectG [2009]. In summary, there are two main motivations behind the design choices taken. First, "DSAL design depends on the component language." This is because join points arise due to events in, or structures of, the base language. Second, "Different problems might be solved in similar domains." In other words, the authors posit that it is the effect that is to be performed by the DSAL that drives the design decisions. The example of AO4BPEL and Padus shows an even stronger result: Even with the same component language and the same problem to be solved design decisions may vary.

In conclusion, various of these decisions remain relevant in the light of more recent work. None of the seven design decisions are, however, tied to the domain that the DSAL is addressing. Furthermore, neither does the goal of the DSAL force a specific choice for all seven.

4.3. Relating the Domain Specificity of the Three Axes

The question that naturally arises is whether there is any relationship between the domain specificity of representation, identification, and effects. Or, does making one of the three axes domain specific necessarily imply making one of the other axes domain specific?

The examples discussed in Section 2 suggest that this is not the case. Indeed, Figure 1 indicates that these three axes may be loosely coupled, as there are examples living at different places in the design space. We now confirm this more systematically by listing four DSALs in Figure 3 that serve as examples for the loose coupling between representation, identification, and effects. This shows that the choice of domain specificity in one of these dimensions does not necessarily determine the specificity of the other dimensions.

Note that a previous attempt at clarifying the relationships between representation, identification, and effects was performed in the summary of the DSAL'08 workshop [Cleenewerck et al. 2008]. That summary, however, is not as complete as what we present here because it does not discuss the relationship between representation and

	REP $\not\rightarrow$ ID	ID $\not\rightarrow$ FX	FX $\not\rightarrow$ REP
DS $\not\rightarrow$ DS	RG	AspectG	PDL
GP $\not\rightarrow$ GP	PDL	RG	AspectG
	ID $\not\rightarrow$ REP	FX $\not\rightarrow$ ID	REP $\not\rightarrow$ FX
DS $\not\rightarrow$ DS	PDL	AspectMatlab	AspectG
GP $\not\rightarrow$ GP	RG	AspectG	PDL

Fig. 3. Examples for loose coupling between the domain specificity of join point representation (REP), join point identification (ID), and effect specification (FX).

	GP	Partially DS	DS
REP	AspectJ	Racer	KALA
ID	AspectJ	Racer	KALA
FX	AspectJ	POM	KALA

Fig. 4. Examples for the different degrees of domain specificity in join point representation (REP), means of identifying join points (ID), and effect language (FX).

effects. Moreover, the domain specificity of all three dimensions is not well defined and consequently some of the examples used do not hold with the definitions we propose here.

In conclusion, a DSAL only needs to provide domain specificity in one of the three dimensions of representation, identification, and effects.

4.4. Degrees of Domain Specificity on the Three Axes

In the previous section, we demonstrated that the three axes of representation, identification, and effects are orthogonal, that is, the choice of domain specificity in one axis does not influence the choice of another. We now show that the three axes are not binary yes/no choices, but that they allow for different degrees of domain specificity. We do this by giving three examples for each of the three axes: one that is purely general purpose, one that is a mix of general-purpose and domain-specific elements, and one that is fully domain specific. Note that we do not wish to imply that the degree of domain specificity of one language is comparable to the degree of another language. Figure 4 enumerates the examples, and we discuss them next.

AspectJ. As the typical GPAL, all three axes are general purpose.

Racer. Racer [Bodden and Havelund 2008] extends the AspectJ representation and pointcut language with three domain-specific join points for concurrent programming: lock, unlock, and maybeShared. The former two represent the acquiring (releasing, respectively) of a lock by entering a synchronized block. maybeShared represents field accesses that may be shared between multiple threads. By adding these join points to the AspectJ join point representation and its pointcut language, Racer increases its domain specificity; however, without removing the general-purpose elements.

POM. Parallel Object Monitors (POM) is a lightweight DSAL for specifying the scheduling of parallel activities in objects [Caromel et al. 2008]. A scheduler is the means to effect at a join point, by suspending and resuming activities. It is written in a general-purpose language (Java), but following a domain-specific framework: Two specific methods have to be implemented and deal with domain abstractions like request queues and priorities. In other words, the domain of effects is restricted to scheduling (although this restriction is not enforced by the implementation).

KALA. Lastly, in KALA (Section 2.3) all three axes have been designed purely to treat elements of the domain. The representation consists of transaction executions and data accesses within them. Identification solely allows for the identification of methods

that correspond to transactions. The effect language only allows for the declarative specification of transactional properties.

In conclusion, we can state that the three axes of a DSAL are more than just orthogonal. Each of these also represents a range of possible choices from general purpose to completely domain specific. As a result, the designer of a DSAL is free to choose the extent of the domain specificity of the language for each of the three axes.

4.5. Means of Identifying and Effecting at Join Points

As we have seen, most GPALs follow the pointcut-advice model first introduced by AspectJ. Some, like HyperJ [Ossher and Tarr 2002], however, do not. Hence, the structure of identification and effects is a decision that should be considered when designing DSALs.

Given that DSLs and hence DSALs are tailored to fit a specific domain, such a tailoring may not necessarily be compatible with the pointcut-advice model. Consequently, a DSAL designer may choose to follow a different approach in which identification is performed and combined with effects. In extreme cases, identification may even be absent in the language. We have, however, only encountered 10 DSALs where the pointcut-advice model is not followed. In chronological order, these are RG [Kiczales et al. 1997], COOL and RIDL [Lopes 1997], AML [Irwin et al. 1997], Robust [Fradet and Südholt 1999], DAJ [Sung and Lieberherr 2002], RemoteJ [Soule et al. 2007], AspectGrid [Sobral and Monteiro 2008], D4OL [Timbermont et al. 2008], and Commentweaver [Horie and Chiba 2010].

We briefly discuss the relevant parts of one example, the COOL language, here. The COOL and RIDL languages form part of the D framework [Lopes 1997], one of the seminal publications on AOP and a precursor to the pointcut-advice model. D was created to deal modularly with selected concerns in the domain of distributed systems through the use of specific DSALs. COOL is a coordination language that provides mutual exclusion of threads, synchronization state, guarded suspension, and notification. RIDL deals with remote method invocation and transferring data between different hosts on the network. In these languages, aspects are called *coordinators* (portals, respectively). In COOL, the join points are method executions, but the identification is very different from pointcuts. Instead of specifying a predicate, the coordinator definition first explicitly mentions the name of the class that is subject to synchronization. Second, it denotes the synchronized methods with their name in `selfex` and `mutex` declarations, which are the means of effecting at these join points. The actual effect is abstracted by these domain-specific operators; the language runtime enforces self- and mutual exclusion of method executions accordingly. RIDL follows a similar design scheme, but we do not detail it here.

In conclusion, DSALs do not necessarily need to follow the pointcut-advice model of identification and effects, yet only one fourth of the languages deviate from this model.

4.6. Domain-Specific Join Points and Optimization

One of the six advantages of DSLs, discussed in Section 3.2, is the ability to perform domain-specific optimization. DSALs add a new kind of possible optimization to the language: along the join point representation axis of the design space, as we discuss here.

In a GPAL, conceptually, all operations of the base program emit a join point, and each of these join points contains a wealth of context information. For example, in a message send join point the context contains both the sender and the receiver as well as the arguments of the message. Aspect languages such as AspectJ perform optimizations as part of their implementation. For example, partial evaluation of the pointcuts can be performed to determine a subset of the program where join points

need to be emitted, and the context information that needs to be reified [Masuhara et al. 2003; Hilsdale and Hugunin 2004].

In a DSAL, optimizations may be domain specific and hence straightforwardly follow from the language specification instead of requiring program analysis. As the representation can correspond to a subset of a general-purpose representation, the implementation of the DSAL may need less join points to be emitted. Similarly, for these join points, the DSAL may require less context information to be present, hence again optimizing join point generation. For example, in KALA (see Section 2.3), there are two types of join points: the execution of a transaction, corresponding to some method executions, and the reading and writing of a transactional value, corresponding to calls to getters and setters within a method that is a transaction. Furthermore, for the transaction execution join point, no context information is present. As a result, the implementation of KALA can (and does) limit join point generation from the outset, without requiring static analysis of the program [Fabry et al. 2009].

Domain-specific optimizations may be more intricate by performing some static analysis combined with domain knowledge. An example of this is Racer [Bodden and Havelund 2008]. Recall that Racer defines a domain-specific maybeShared pointcut (and its corresponding join point) that matches all field accesses that may be shared by multiple threads. A naive implementation of this is to match on all field accesses. Instead of this, a domain-specific optimization is performed using a complex thread-local objects analysis algorithm. In general, this permits one to greatly reduce the number of maybe-shared join points.

Apart from Racer and KALA, the only other mention of domain-specific optimizations in the join point model we are aware of is in the work on Sequential Object Monitors (SOM) [Caromel et al. 2004]. SOM is implemented in the Reflex AOP Kernel [Tanter and Noyé 2005], discussed in Section 5.5.3. The customization of the join point representation to avoid embedding all available context information is one of the keys to the good performance of SOM [Tanter 2004].

In conclusion, a number of domain-specific optimizations may be performed by a DSAL, but only few of the studied languages perform them. A possible explanation for the lack of domain-specific optimizations is that the proposed languages do not yet have performance requirements.

4.7. Composition of DSALs

Our study revealed various instances of a setting where more than one DSAL is expected to be used when programming an application. This does not come as a surprise, since in a large application multiple aspects may be present. Hence, when each of these is written in a different DSAL, multiple DSALs must cooperate. The examples of composition of different DSALs we encountered are the combination of COOL and RIDL, in the setting of Lopes's PhD thesis [1997], and the family of DSALs that has been built on top of KALA [Fabry et al. 2008]. Also, when considering DSAL implementation toolkits, we find that 7 of the 36 we surveyed provide support for composition (we discuss this in Section 5.5).

Hence, a decision that needs to be made when designing a DSAL is whether to consider the composition of multiple DSALs, that is, how the language enables the composition of programs written in different DSALs, together with the base application, to form a correct application. In the of supporting some form of composition, there are two distinct options that can be chosen: first, different DSALs can be codesigned; and second, a DSAL might be designed explicitly taking composition with other, unspecified, DSALs into account. Moreover, for the latter case a number of DSAL infrastructures have been created with support for composition built in.

4.7.1. Codesigned DSALs. The example languages we encountered all fit in the first category: They are cases where different DSALs have been codesigned, and this because the different concerns they each address fit within a larger domain. For example, both COOL and RIDL have been developed as part of the work on D, a framework for distributed programming [Lopes 1997]. As the domain of distributed programming is quite extensive, various parts of it can be addressed by multiple languages, and each of these languages still addresses a more or less self-contained domain. In this case, RIDL addresses data transfers between different execution spaces and COOL addresses concurrency, both of which arise due to remote invocations. Consequently, we have two DSALs that have been codesigned, such that each of them addresses a part of the domain of distributed programming. Considering the composition of COOL and RIDL code, this composition is easy because there is a simple relationship in the functionality of the two languages. Both act on method invocations: RIDL concerns itself with passing the parameters on remote invocations, while COOL treats the execution of the methods themselves. As these are two different phases in the invocation process of the method, there is no overlap in functionality and the composition is straightforward.

From the examples, we established that when codesigning different DSALs, the language designer may consider at least three points. First is the organization of the different concerns: which structure or behavior is part of which concern and how do the different concerns compose, as shown for COOL and RIDL earlier. Second is the definition of a common vocabulary for shared concepts, such that they are referred to in the same way across the different languages. In the aforementioned case, the common vocabulary is simply methods and classes. Third is the definition of a standard syntax for the shared vocabulary, to ease the learning curve of the different languages. In the previous example, classes and methods are identified in the same way, by giving their name and signature, respectively.

4.7.2. DSALs with Support for Composition. We have not found any examples of the second category of DSALs designed for composition: those that explicitly take composition with other DSALs into account. Nonetheless, a language designer should be aware of the issues of composition and interactions of DSALs. Such issues have been raised by the community in the research on implementation toolkits. In particular, Lorenz and Kojarski [2007] introduced the notions of coadvising and foreign advising. Coadvising refers to the composition of effects from multiple aspect languages at a shared join point. Foreign advising refers to the fact that one aspect may react to a join point related to the execution of the effect of another aspect written in another aspect language.

In the case of coadvising, the order in which the effects of the different aspects will take place may influence relevant behavior of the application. A DSAL designer may want to explicitly state what the requirements are in the execution of its effects to respect the intended semantics of the language. For example, a hypothetical DSAL for real-time systems may implement a log of the execution times of methods, which should include the behavior of aspects that apply at method executions. In this case, the designer could state that, when coadvising, the start timer effect should always take place first, and the stop effect last.

In the case of foreign advising, on the one hand, the DSAL being written may emit join points that are of interest to other DSALs. On the other hand, it may act at a join point emitted by effects of another DSAL. For the former case, the DSAL designer should therefore consider the repercussions of the behavior of (parts of) the programs in the language being interrupted by other aspects, and may want to document restrictions, if there are any. For example, considering the hypothetical DSAL for real-time systems, it may contain sections of behavior that need to respect constraints in their execution time and, therefore, may not be interrupted by other aspects. For the latter case, the

designer should consider if the effects of other (known) aspects should be observed by DSAL programs and document this.

4.7.3. Composition Support in Toolkits. As already mentioned, a significant amount of work on the composition of DSALs has actually taken place in the context of the work on DSAL implementation toolkits. The most effort with regard to providing support for composition as part of the DSAL toolkit is the ongoing work of Lorenz et al., in the context of the Awesome implementation toolkit [Kojarski and Lorenz 2007]. The goal of this work is to obtain what they term language-oriented modularity: "the process of constructing and composing DSALs to better support aspect-oriented modularity" [Lorenz 2012]. This work originated with Pluggable AOP [Kojarski and Lorenz 2005], where the aspect extension composition question is defined and studied. Briefly put, this question is about what the semantics are of a base program in the presence of multiple aspect programs, written in different aspect extensions, that is, multiple DSALs. Pluggable AOP proposes a method in which the base and different DSALs may be implemented such that they collaborate to yield a semantics of the composition that can be derived from the semantics of the constituent parts. Awesome can be seen as a follow-up of Pluggable AOP: a composition framework "for constructing a multi-extension weaver by plugging together independently developed aspect mechanisms" [Kojarski and Lorenz 2007]. The framework also provides for a means to customize composition behavior. This has been used in more recent work [Lorenz and Mishali 2012] to provide support for declarative specifications of the composition, as well as for each of the different DSALs. Finally, integration of multiple DSALs using one toolkit also provides for debugging support that takes the different languages into account. This is performed through a specific multi-DSAL debug interface [Apter et al. 2012] that reveals the different effects that apply at a given point, as specified by the different DSALs. Additionally, as an explicit composition specification is available, this is revealed by the interface such that it can be studied by the programmer as part of the debugging process.

In conclusion, all cases of DSALs with support for composition have been codesigned and there are specific points to be taken into account when doing this. For a language that is not codesigned with another DSAL, the work on toolkits has also provided concepts and tools that are relevant for its design. This as a significant amount of work on DSAL composition has been performed in that area.

5. DSAL IMPLEMENTATION TAXONOMY

When DSALs are implemented, a DSAL developer needs to select the most appropriate language approach for this implementation. However, the selection of the right approach may not be easy for the developer because the literature gives developers only little orientation and recommendation on how to develop a DSAL. Therefore, this section proposes a taxonomy of implementation patterns for DSALs. DSAL developers can use this taxonomy as a catalogue to select the most appropriate way to implement their languages.

Since DSALs are DSLs, the implementation of a DSAL has fundamental similarities with the task of implementing a DSL and it would be natural for DSAL developers to consult taxonomies for DSL implementation. Mernik et al. [2005] propose a taxonomy of implementation patterns for DSLs, but do not consider DSALs.

At first glance, it may seem that DSAL developers may simply follow the recommendations of Mernik et al. However, it turns out that those recommendations are not completely transferable to DSAL implementation because there are additional implications and issues when implementing a DSAL. The most notable of these are the concerns about the complexity of DSAL implementation, reuse, and performance:

—In addition to DSLs, DSAL implementations are faced with the *complexity* of having to support aspect-oriented composition mechanisms.
—DSALs have important opportunities for *reuse*, since often a DSAL developer only needs to slightly adapt part of an existing aspect-oriented composition mechanism for the particular domain.
—DSALs have additional *performance* issues compared to a typical DSL because join point reification is potentially costly.

We, therefore, elaborate in this section a taxonomy of *DSAL implementation patterns* that prescribe the architecture of a DSAL implementation. Our DSAL taxonomy is inspired by the classification of DSL implementation patterns from Mernik et al. and uses similar classes of patterns: (1) *interpreters*, (2) *compilers*, (3) *embedding*, and (4) *hybrids*. In our classification, we unify the *compiler* and *preprocessor* classes in a single *compiler* class, as we consider preprocessors simply as one way among others to provide a DSAL compiler. Moreover, we do not consider a *commercial-off-the-shelf* (COTS) class, defined by Mernik et al. as an approach that "builds a DSL around existing tools and notations," since reuse of existing infrastructure arguably already happens in all the other classes. Last, we also include one extra class: (5) *DSAL infrastructures*, which provide specific support for implementing DSALs.

In the following subsections, one for each class, we define the DSAL implementation pattern, mention existing DSAL implementations that use the corresponding pattern, and summarize one representative DSAL implementation of the corresponding class. The issues of complexity, reuse, and performance differ strongly in each implementation pattern, and therefore we also elaborate on the peculiarities of each pattern with respect to these issues. To end this section, we also provide a classification of the languages we studied, according to their implementation pattern.

5.1. DSAL Interpreters

A *DSAL interpreter* executes DSAL constructs directly after syntax analysis. Only two out of the 36 DSALs use the interpreter pattern in their implementation. A DSAL interpreter implementation can be divided into the following parts:

(1) *Base Part*. The base part is responsible for evaluating base language constructs in a given environment.
(2) *Instrumentation*. The instrumentation reifies join points, which exposes part of the base environment to the aspects.
(3) *Aspect Part*. The aspect part is responsible for evaluating DSAL constructs.
(4) *Integration*. By composing all parts, a DSAL developer creates a common interpreter environment such that this common intepreter reifies join points, matches pointcuts, evaluates the corresponding advice, and then reflects changes back into the base environment.

There are three main approaches to build a DSAL interpreter: first, implementing a DSAL interpreter from scratch; second, performing an adaptation of an existing base interpreter that was not designed for extension; and third, extending an extensible base interpreter. Of the two DSAL interpreters for the languages we studied, none have been implemented from scratch. One interpreter has been implemented as an adaptation, and the other as an extension. We provide remarkable properties of the following three approaches.

First, when implementing a DSAL interpreter from scratch, the developer can encode the aforementioned four parts in one homogeneous system. The advantage is that the resulting system can easily integrate domain-specific aspect-oriented constructs with

the interpretation logic of the base constructs, since the developers have great flexibility for structuring the implementation.

Second, when a base interpeter is available without special support for extensions, the DSAL developer can still adapt the source code of such a base interpreter for supporting domain-specific aspects. Implementing a DSAL interpreter by adapting an existing DSL or GPL base interpreter is arguably one of the simplest techniques to DSAL implementation. However, there is the implementation level issue that the resulting implementation of the aforementioned four parts may be tangled in the source code of the interpreter.

Third, using an extensible base interpreter addresses the tangling problem if it allows one to reify join points and perform the join points effects by a (modular) extension. The developer can then define the three remaining parts of the DSAL as modular extensions to the base interpreter. The advantage is that large parts of the base interpreter implementation may be reused in the DSAL extension. Furthermore, the base and aspect parts can be independently updated. Because they are decoupled, only when a base language construct relevant to DSAL semantics is added do the aspect-oriented parts need to be updated as well.

5.1.1. The Pluggable AOP Interpreter. The first published work on using an interpreter-based DSAL approach is by Kojarski and Lorenz [2004], called Pluggable AOP and used to implement a simple GPAL and the COOL DSAL in Scheme. Being the first work, it set a reference point for the interpreter-based approach, and hence we discuss it here. A key property of the approach is that it enables DSAL developers to reuse the same base interpreter implementation with different DSAL extensions and also allows their composition. All interpreters are structured in a *pipes-and-filters* architecture. Each of them implements an eval function that interprets all language constructs of interest and delegates the evaluation of the other constructs to the interpreters that come later in the pipeline. While the base interpreter evaluates only base language constructs with default execution semantics, aspect interpreters evaluate aspect-oriented language constructs as well as (possibly) base constructs. The latter happens when these fall under aspect-oriented semantics that match pointcut and execute advice. Because aspect language interpreters are positioned earlier in the pipeline than the base interpreter, which is the last in the pipeline, an aspect interpreter can intercept and change the evaluation of base constructs.

5.1.2. Tradeoffs with Interpreters. There are a number of advantages, but there are also important limitations to using the interpreter pattern.

Since interpreters allow for a rather easy and rapid implementation compared to compiler approaches, DSAL developers may want to use this pattern for prototyping DSALs. Because most interpreters allow direct access to the dynamic context of the DSL constructs, interpreters are an easy way to implement DSALs with a rather dynamic nature. Another advantage of accessing the dynamic context in the interpreter pattern is that it enables a more flexible control over the execution of base and DSAL constructs than in other patterns.

However, DSAL interpreters are limited by their low performance in comparison to the other patterns. This is because of two main reasons: First, interpreters have the usual interpretative overhead. Second, since interpreters dynamically compose aspects, one can expect a rather large overhead due to this runtime weaving process.

Usually, a DSAL interpreter dynamically reifies every join point during the evaluation of base constructs and passes them as objects to the aspect part, which then performs pointcut evaluation and advice execution. It is likely that the execution performance of a DSAL interpreter will only be acceptable when the DSAL interpreter

reifies a relatively small portion of join points, or when the interpreter performs partial evaluation of pointcuts.

Because of the mentioned tradeoffs, we recommend the DSAL interpreter pattern only if there are no hard performance requirements, as well as for prototyping DSALs.

5.2. Compilers

DSAL compilers compile both a base program and a DSAL program in order to produce a woven program in a certain target language. Such a compiler can be structured into a part that takes care of the semantics of base language constructs and a part for the DSAL constructs, but in contrast to interpreters, a DSAL compiler may also perform semantic analysis without executing the source code.

It may seem obvious that DSALs can be implemented like ordinary GPL compilers or DSL compilers. However, the design of a DSAL compiler strongly depends on the existing design and implementation of the compiler of the base language. This is because the DSAL part of the compiler needs to be fully integrated with the base part to work. 25 of the 36 DSALs we studied are implemented using a compiler approach. Only one of them implemented the compiler from scratch (and two are unspecified). The large majority of DSALs hence are implemented by extending the existing base compiler.

When there is a special extension mechanism available in the base compiler, that is, it is an extensible compiler [Nystrom et al. 2003; Ekman and Hedin 2007], the DSAL compiler can be realized as such an (preplanned) extension. Alternatively, developers implement DSAL compilers as ad hoc extensions to their base language compiler. We briefly discuss these two categories—*ad hoc compiler adaptation* and *planned extensions*—next.

Ad hoc Compiler Adaptation (20 out of 22 compiler adaptations). A DSAL extension can build on an existing base compiler even if no special extension mechanism is available in the base compiler implementation. Conceptually, the simplest adaptation consists of adding a preprocessing phase to the base compiler (17 out of the above 20). According to Mernik et al. [2005], a preprocessor is characterized by the fact that "DSL constructs are translated to constructs in an existing language" and "Static analysis is limited to that done by [the compiler]." We have found three distinct classes of use of a preprocessor: first, to produce woven code in the base language (9 out of 17); second, to translate to a GPAL for the base language (6 out of 17); and third, to generate woven code in a language different from the base (2 out of 17). An example of the third case is PDL [Morgan et al. 2007] (see also Section 2.2), where C# code is generated, and the base is .Net assemblies. Remarkably, the PDL preprocessor does perform type checking and optimization. In our survey, we, therefore, need to broaden the preprocessor definition of Mernik et al. to remove the restriction placed on where static analysis occurs.

Alternatively to preprocessors, when the technology of the base compiler lacks extension mechanisms, the DSAL developers may choose to invasively adapt the code of the base compiler (if available). The quality of such an adaptation completely depends on the design of the compiler being adapted: how well its internal structure allows extensions and so on.

Planned Extension (2 out of 22 compiler adaptations). A particularly attractive approach for implementing DSALs is to use an extensible base language compiler [Nystrom et al. 2003; Ekman and Hedin 2007]. Using an extensible compiler as a base compiler is attractive, as it provides specific extension points for adapting the compiler implementation. In particular, the parser can be extended for recognizing DSAL constructs, and the semantic analysis and transformation phases can be extended for weaving aspects. Thanks to reusing large parts of the standard compilation logic for

the base language, significant savings can be expected when implementing a DSAL extension on top of it.

To the best of our knowledge, there are two aspect compilers based on an extensible base compiler: the compiler for Conspects [Holzer et al. 2011] and the compiler for AspectMatlab [Aslam et al. 2010]. The latter is particularly interesting as it is an aspect compiler for a domain-specific base language; therefore, we discuss it in the following text.

5.2.1. The AspectMatlab Compiler. AspectMatlab is an aspect extension for Natlab (Neat Matlab), a reduced version of the Matlab language. AspectMatlab is implemented by performing a modular extension to the McLab extensible Matlab compiler [Casey et al. 2010].

The AspectMatlab extensions are built using the language-independent extensible compiler framework JastAdd [Ekman and Hedin 2007], best known for JastAddJ—its Java port. To enable the aspect-oriented syntax for Matlab, these extensions extend the grammar of Natlab to incorporate the additional AspectMatlab productions. For enabling the aspect-oriented semantics, AspectMatlab uses JastAdd to implement several transformations that traverse the Abstract Syntax Tree (AST), associate attributes with AST nodes, and modify the AST by rewriting nodes.

Compilation consists of several transformation steps that are performed after parsing. The first step simplifies expressions to transform the join point shadows of composite Matlab expressions to a normal form. This simplifies the transformations for the weaving later on. The second step enables weaving on loops, by rewriting them such that increment steps in for-loops and conditional expressions of while-loops become visible and modifiable for aspects. The third step performs a name-resolution analysis to resolve the types of used identifiers to enable optimized weaving. The fourth step is the matcher and weaver transformation. It evaluates pointcuts, using static information from the AST and the types from the previous step. When insufficient static information is present, or the pointcut uses dynamic information, a dynamic condition is woven in. In the fifth and last step, the AspectMatlab compiler produces woven Matlab source files.

5.2.2. Tradeoffs with Compilers. Compared to interpreters, compilers perform syntactic and semantic analysis before program execution. This has two main benefits: robustness and performance. This can be particularly effective in the case of a DSL as program analysis can be tuned to take into account the specificity of the language. However, most compiler approaches in the literature have not been designed with aspect language development in mind. As a result, developers have little support for implementing the weaving process, which may be complex and hence make up a significant proportion of the cost of implementing a DSAL. Nonetheless, 25 out of the 36 DSALs we studied are implemented as compilers, most of them by using a preprocessor.

The concrete set of requirements for a DSAL determine what compiler implementation approach is best. We elaborate on this in the following paragraphs.

DSAL Compiler from Scratch. When implementing a compiler from scratch, DSAL developers have great flexibility for implementing special logic for the DSAL semantics. However, in contrast to other implementation patterns, there are significantly large initial costs for implementing a complete compiler infrastructure, which is well known for GPL and DSL compilers in general [Kosar et al. 2008]. Another problem is that developers have little guidance when starting from scratch. Therefore, we recommend this approach only for developers that have experience with implementing compilers, that is, for compiler experts. If there are no specific requirements, a DSAL would be better implemented with a compiler framework or extensible compiler, as the developer can expect lower initial costs.

Ad hoc through Preprocessors. There are limitations of the preprocessor approach due to the fact that join point reification and weaving are implemented only as a syntactic analysis. In case the target language is a GPL, code produced by the preprocessor must be compilable with the GPL compiler. Therefore, only implementing source code weaving is supported, in contrast to weaving through semantic analysis. In case the target language is a GPAL, DSAL aspects can be considered to be only syntactic sugar for GPAL aspects. Therefore, the preprocessing approach may still yield a powerful DSAL with efficient code. Because of the known limitations, we recommend preprocessors only if the developer knows that the syntax can be rewritten without complex transformations or semantic analysis.

Alternative ad hoc Adaptation. Because of the diversity in compiler implementations, there are few recommendations that can be given to the code adaptation approach in general. We advise against such ad hoc compiler adaptation, unless the compiler has a very clear structure and implementation, or the adaptations are trivial. This is because of the high complexity that such an adaptation may entail. However, this approach may be the only available option in case there is no extensible base compiler.

Planned Extensions. The extensible compiler approach is best suited for DSALs that are extensions to a base language for which an extensible compiler is available. However, for most base languages, regardless of whether they are general purpose or domain specific, there is no extensible compiler available. Therefore, implementing a new extensible compiler approach for DSALs is only recommended after calculating the tradeoff between the investment for the infrastructure and the payoff in savings of implementing the DSAL extensions.

5.3. Embedding

In embedding approaches [Hudak 1996], a developer embeds a DSAL by implementing DSAL abstractions as a library in an existing *host language*, typically a GPL. Different models of embedding exist: homogenous embedding [Hudak 1996], where the system of the host language also takes care of the embedded language; heterogeneous embeddings [Tratt 2008], where the embedded language is handled by a separate system; and polymorphic embedding [Hofer et al. 2008], which focuses on providing different possible implementations of the DSL. Renggli et al. [2010] further classify the embedded languages in Pidgins, Creoles, and Argots, depending on how syntax and semantics is adapted. However, none of the DSALs we studied have been implemented as an embedding. The work on DSAL embeddings has so far been contained to developing the approach using smaller example languages and, as mentioned in Section 3.3, these smaller languages are not part of our study.

For the embedding approach in general, the host language is required to provide certain special host language features that allow for directly implementing DSAL abstractions and the corresponding weaving logic for DSAL aspects within a library of that language. We consider three classes of host languages: (1) aspect-oriented languages, where the DSAL inherits the base language of the host language; (2) reflective languages, where the base language of the DSAL is, often implicitly, the host language without reflection (otherwise mixing aspects and reflection may be hard to control); and (3) GPLs, which may also act as the base language. We discuss these three options next.

The first possibility is to use an aspect-oriented host language for implementing domain-specific aspects. The host language typically is a GPAL, such as AspectJ. The developer uses the GPAL abstractions of this language to define a DSAL library with domain-specific join point identification and domain-specific effects. Examples of this are existing embedded aspect libraries for certain domains (e.g., for design patterns [Hannemann and Kiczales 2002; Noble et al. 2007]). Such libraries define more or less abstract DSAL aspects using GPAL aspects. These DSAL aspects come with predefined

pointcuts and advice. The (abstract) pointcuts define the vocabulary, while the advice implements concern logic for the domain.

The second possibility is to use a powerful host language with reflective language features for enabling the implementation of weaving. There are two important flavors of using reflection for implementing GPALs. On the one hand, there are *metaobject protocols* (MOPs) [Kiczales et al. 1991] in which aspect semantics are mapped to metaobjects that adapt the default execution of base objects [Bergmans and Aksit 2001; Hirschfeld 2003; Dinkelaker et al. 2010; Achenbach and Ostermann 2010a]. On the other hand, there is the use of *metaprogramming*, where aspects are mapped to metaprograms that insert advice logic at relevant join points [De Volder and D'Hondt 1999; Baker and Hsieh 2002; Wampler 2008]. MOPs can be used for implementing DSALs; for instance, in Dinkelaker et al. [2010] and Achenbach and Ostermann [2010a, 2010b]. Alternatively, when special aspect semantics are required in a certain domain, one can use the metalevel for aspects—called a *metaaspect protocol* [Dinkelaker et al. 2009; Achenbach and Ostermann 2010a]. Similar to metaobjects, in a metaaspect protocol, metaaspects can adapt the default execution of aspects for implementing application-specific aspectual semantics, such as debugging semantics for pointcuts and context-specific composition of advice.

The third possibility is to use an available indirection mechanism. This, however, usually supports only a limited set of aspect-oriented constructs; typically only permitting before and after advice, not around advice. For instance, dynamic proxies in Java are used by general-purpose frameworks like Spring AOP [Walls and Breidenbach 2005] and JBoss AOP [Fleury and Reverbel 2003]. There are at least two proposals for DSALs that make use of dynamic proxies. Sobral et al. [2006] use proxies for implementing *parallelization aspects*, involving *object partitioning*, *concurrency management*, and *distribution*. More recently, Soule [2010] used JBoss AOP for implementing autonomic computing concerns.

5.3.1. The Reflective Embedding Approach. As an example, developing the embedding approach, consider the work of Dinkelaker et al. [2010]. This work proposes an architecture for the realization of such an implementation that addresses composability of different DSALs as well as providing ease of implementation, which is why we discuss it here.

To implement new DSALs with reduced development costs and without requiring complicated compiler techniques, Dinkelaker et al.embed aspect-oriented concepts together with domain-specific ones into Groovy. They demonstrate the approach by implementing a security DSAL for a small workflow language embedded in Groovy. The DSAL implementation process starts by embedding the domain-specific workflow language, that is, the base language. Next, to enable a base embedding for aspects, the reflective features of the host language are used to implement the reification logic. This logic extracts domain-specific join points and their execution context from the base embedding. To enable join point identification and using domain-specific effects at these join points, the developer implements a domain-specific join point identification language and effect language. Finally, these embeddings are passed to an aspect-oriented kernel that is a generic aspect-oriented weaver also embedded into the host, which then assembles these parts to form a DSAL weaver. This way, the approach allows for implementing a rich set of aspect-oriented language features (e.g., dynamic join points and aspects, without requiring that the DSAL developer knows the details of how to implement an aspect-oriented weaver).

5.3.2. Tradeoffs with Embedding. An advantage of embedding DSLs is low implementation costs [Kosar et al. 2008]. In the case of a DSAL, the implementation of the aspect weaver is simplified. Sticking to the syntax and semantics of the host language,

however, imposes the constraint of using the same granularity of join points as that of the host language. Finally, embedded DSLs tend to rely on the general-purpose program analyses provided by the host language, which cannot eliminate the complete overhead of all the library calls resulting from the embedding. This gives an interpretive flavor to the approach.

To summarize, we only recommend the embedding approach if there is no requirement for a concrete DSAL syntax and there are no high-performance requirements.

5.4. Hybrids

Hybrid approaches combine some of the previous approaches in order to alleviate their individual drawbacks. Of the 36 languages we studied, one is implemented as a hybrid. When also considering the implementation approaches, we found two kinds of hybrid approaches.

The first kind combines preprocessing with interpretation or with some form of embedding. The one example language we found, AOWP-PHP [Hokamura et al. 2008], uses preprocessing to insert a join point creation process (see Section 3.1.1) in the base code, combined with an interpreter for the aspects. When considering embedding, preprocessing may apply to the base program as in TwisteR [Achenbach and Ostermann 2010a, 2010b]), advice as in Dinkelaker and Mezini [2008], or both, as, for instance, in Reflex [Toledo and Tanter 2008] or JAMI [Havinga et al. 2008]. This is to alleviate the drawbacks of the embedding approach discussed in Section 5.3. In the case of TwisteR, the approach allows the implementation of join point creation at different levels of granularity. TwisteR is discussed in more detail in Section 5.4.1. In the other cases, the approach bridges the gap between the API provided by the embedding and a user-friendly concrete syntax.

The second approach combines embedding using a GPAL, typically AspectJ, with some form of interpretation as in Dinkelaker and Mezini [2008], CALI [Assaf and Noyé 2008], or JAMI [Havinga et al. 2008]. The aim is to reuse an efficient compiler-based weaver and complement it with an interpretation layer providing extensibility. This interpretation layer can be at the advice level [Dinkelaker and Mezini 2008] or between the join points generated by the GPAL and domain-specific advice (CALI and JAMI). In the second case, this realizes a form of two-step weaving [David et al. 2001]: The GPAL provides a first step of compiled weaving, which, at runtime, generates join points triggering, as their advice, an interpretation layer that performs a second step of runtime weaving, interpreting the initial join point as domain-specific join points and scheduling the appropriate effects. This second step of runtime weaving goes much beyond the mere evaluation of residues performed by AspectJ [Hilsdale and Hugunin 2004]. This is further discussed in Section 5.5.1.

Both approaches can also be used together as in Dinkelaker and Mezini [2008] and JAMI.

5.4.1. TwisteR. TwisteR is a meta-aspect protocol, embedded in Ruby, which was specifically designed to cover the requirements of dynamic analyses in dynamically typed languages [Achenbach and Ostermann 2010a, 2010b]). In this setting, various kinds of join points are useful. Whereas standard join points, as provided by AspectJ, are enough for analyzing performance and memory consumption, fine-grained join points at the levels of basic blocks or even primitive statements such as assignments are necessary for advanced monitoring and debugging tools. The join point creation support of TwisteR is quite exceptional, which is why we discuss it here.

In order to provide, depending on the analysis, the appropriate join points a preprocessor rewrites the program AST, injecting code responsible for join-point reification. Join-point reification is then available for any AST node, including basic blocks,

control-flow constructs, and assignments. The embedded part of the DSAL implementation is responsible for analysis, performed by domain-specific aspects. On the one hand, a pointcut defines which join points in a program are relevant for the analysis; on the other hand, the corresponding advice updates the analysis result.

Achenbach and Ostermann demonstrate TwisteR by implementing two DSALs for analysis in Ruby, which is used as both the base and the host language. The first DSAL is used for exploratory testing with trivalent logic and the other DSAL is used for debugging.

5.4.2. Tradeoffs with Hybrids. The tradeoffs of the hybrid approach are mostly determined by what DSAL approaches developers combine, what the tradeoffs between these approaches are, and how developers combine the approaches. There can be both positive and negative effects with such a combination.

First, when combining embedding with preprocessing, the preprocessing step can leverage more flexibility as seen with TwisteR preprocessors can solve the general problem of the lack of concrete syntax of an embedded DSL [Bravenboer and Visser 2004; Dinkelaker et al. 2013]. However, introducing a preprocessing step can destroy a possible homogeneous nature of the embedding, which may pose problems when composing several DSALs. This has, for instance, been experienced with Reflex (see Section 5.5.3).

Second, combining a DSAL interpreter with a compiler enables faster execution than with a pure DSAL interpreter. In such a setting, the compiler provides the semantics for the nonextensible features of the language, which run fast, and the interpreter provides the semantics of the extensible ones. However, the overall interpretation of aspects can be expected to be still slower than with a compiler completely implementing the semantics for all language features of the DSAL.

Third, when using preprocessors to make additional information visible to a DSAL, additional indirections may be created for new join points, resulting in an additional performance overhead. When composing preprocessing phases, care must also be taken not to erroneously advise this synthetic code.

All in all, using a hybrid approach requires one to master the various individual approaches that are combined as well as their combination. It is inherently more complex, although it also makes it possible to reuse existing building blocks. In conclusion, we recommend one carefully consider the advantages and drawbacks of such an approach before using it in the case of a single DSAL. However, in the case of a family of DSALs, this may be a very interesting approach, providing a good mix of efficiency and flexibility.

5.5. DSAL Infrastructures

DSAL infrastructures help developers implement new DSALs by providing processes, frameworks, libraries, and tools (like generators) to facilitate this work. We here give an overview of the different infrastructures we encountered in the literature.[2] Only five of the 36 DSALs we studied were implemented using these DSAL infrastructures.

5.5.1. Java Aspect Metamodel Interpreter. The Java Aspect Metamodel Interpreter (JAMI) [Havinga et al. 2008] is an aspect interpreter framework that focuses on implementing and composing DSALs. JAMI implements a high-level structure for common aspect-oriented concepts as a set of classes in a metamodel framework. These classes

[2]We do not include the Aspect SandBox (ASB) [Masuhara et al. 2003] and Extended-Aspect SandBox (X-ASB) [Ubayashi et al. 2004] as they are not designed as a DSAL infrastructure. Instead, their goal is to provide a prototype environment for experimenting with AOP semantics and implementation techniques.

implement the composition semantics that matches pointcuts and then orders and executes advice.

Implementing a DSAL in JAMI entails instantiating the framework. If necessary, the standard framework semantics can be adapted by subclassing the metamodel classes. After DSALs have been implemented, developers can compose them such that aspects written in different DSALs can advise the base code. To compose DSALs, the developer implements a combined parser of the DSALs and integrates the different framework extensions into one extension. JAMI has been illustrated by implementing COOL and composing it with a DSAL for caching method return values [Havinga et al. 2008].

To resolve DSAL composition conflicts, DSAL developers can extend the standard composition semantics so that the aspects of different DSALs are woven in a special way. The developer can control coadvising conflicts (see Section 4.7.2) at shared join points between aspects of different DSALs (e.g., they can prevent that COOL aspects conflict with caching aspects). To ensure a correct ordering of such DSAL aspects, the developer needs to provide code that adapts the framework extension points to resolve all coadvising conflicts.

JAMI is good for incremental DSAL development, since developers use OO inheritance for extensions, and it can help controlling complicated aspect interactions between DSALs at shared join points.

5.5.2. Aspect-Bench Compiler. The Aspect-Bench Compiler (abc) [Avgustinov et al. 2006] is an extensible aspect-oriented compiler framework that specifically targets implementing aspect-oriented languages for Java. There are two versions of abc, the older version using the Polyglot [Nystrom et al. 2003] extensible Java compiler, and the newer version using the JastAddJ [Ekman and Hedin 2007] extensible Java compiler. In essence, abc is an extensible AspectJ aspect weaver. Hence, DSALs that are extensions to AspectJ can be implemented using abc with a lower amount of work than implementing a new weaver. This is because, for these languages, developers only need to provide an extension module that defines the new join point identification and effects. The abc compiler has successfully been used to implement various extensions to Java and AspectJ [Bodden 2005; Bodden and Havelund 2008; Avgustinov et al. 2008; Nusayr and Cook 2009; Marot and Wuyts 2008]. As another example, Racer [Bodden and Havelund 2008] (discussed in Sections 4.4 and 4.6), is also implemented using abc.

The abc compiler is designed to support easy extensibility. Its support for attribute grammars enables mixins and the automatic scheduling of calculating attributes, which helps developers implementing complicated extensions and resolving dependencies between parts of them. Moreover, using abc results in DSALs with a performance suited for production environments. Unfortunately, abc is not designed to handle composition conflicts between multiple DSALs [Havinga et al. 2008].

5.5.3. Reflex. Reflex is a versatile kernel for defining aspect-oriented programming mechanisms on top of Java [Tanter and Noyé 2005]. At its core lies a model of partial reflection [Tanter et al. 2003], which makes it possible to reify both behavioral and structural elements of Java programs. Reflex provides an intermediate abstraction between high-level, possibly domain-specific, aspect language elements, and low-level bytecode transformation. A composition layer makes it possible to handle interactions at this intermediate level [Tanter 2006; Tanter and Fabry 2009]. To facilitate the implementation of aspect languages, Reflex integrates the MetaBorg [Bravenboer and Visser 2004] approach for unrestricted embedding and assimilation of domain-specific languages. Concretely, MetaBorg provides the extensible concrete syntax layer that Reflex language plugins use to generate Reflex configurations.

The ReLAx case study: A reimplementation of KALA (see Section 2.3) on top of ReflexBorg was the first experiment of a full-fledged domain-specific aspect language with this architecture [Fabry et al. 2009].

The ReflexBorg approach has been shown to be practical in the different experiments taken in isolation. But the fact that several transformation layers are involved makes composition issues, especially between languages, hard to deal with. As there is only basic support for backward traceability in the core (in order to be able to trace back conflicting intermediate abstractions to their original source elements), the use of ReflexBorg in complex scenarios is more difficult.

5.5.4. Awesome. Awesome [Kojarski and Lorenz 2007] is a multiweaver approach for controlling interactions of DSALs, already mentioned in Section 4.7. Internally, Awesome extends the ajc compiler [Hilsdale and Hugunin 2004] and, as a result, uses the AspectJ join point representation. The Awesome compiler implements a four-step weaving process. The first step, called *reify*, defines how join points are created and how their context is extracted from Java. The second step, called *match*, implements semantics for join point matching. The third step, called *order*, implements the semantics for advice ordering. The fourth step, called *mix*, weaves the advice logic into the base system. While the first and fourth steps are common and provided by the platform, the second and third steps are extension points at which the developer can provide DSAL extensions.

To implement a DSAL, a developer needs to implement a front end and a back end. The front end implements the parser for the DSAL extension. The back end extends the weaver process by providing the match and order steps. On the one hand, the match step matches pointcuts of this DSAL extension in isolation. On the other hand, the order step orders advice from that extension with respect to other advice of the same extension. To compose DSALs, the developer can again extend the match and the order steps to resolve coadvising and foreign advising interactions. Recently, an additional tool called *Spectackle* [Lorenz and Mishali 2012] proposed that helps with detecting and resolving conflicts by means of declarative specifications of the compositions.

Awesome has been used to implement a COOL variant called *CoolAJ* [Kojarski and Lorenz 2007]. CoolAJ has been composed with AspectJ, resolving coadvising and foreign advising interactions by correctly ordering the advice from the two extensions.

The Awesome compiler is good for DSALs with performance requirements, since compiled DSAL programs yield a performance comparable to statically woven GPAL programs.

5.5.5. The Aspect Markup Language. Lopes and Ngo [2004] propose a language-agnostic preprocessor approach called the Aspect Markup Language (AML). Lopes et al. criticize the dependence on the Java platform as well as the general-purpose nature of AspectJ. The latter is criticized because it implies that programmers must use its generic and relatively low-level model of crosscutting. Instead, aspect languages should empower the domain experts to have direct control over the software by providing more declarative language features for their domain.

To address the two problems just mentioned, AML builds on the extensibility of an XML-based language infrastructure. AML specifies aspects as pointcut and advice pairs in XML syntax and these then become executable by mapping them to a particular platform. The mappings are implemented as generic transformations from XML to platform-specific GPAL aspects. For example, a mapping to AspectJ is provided.

To define a declarative aspect language in AML, a DSAL developer creates a plug-in module that augments the core of AML for the respective domain-specific concern. More specifically, each plug-in extends the core XML syntax and implements a code

generator for the domain-specific details. As examples the implementation of a DSAL for tracing and another for design patterns are shown.

AML implements aspect weaving using a two-phase weaving process. First, when parsing an AML aspect, XML elements that represent syntax extensions are recognized by the XML parser. The parser then delegates the extraction of the information relevant for a domain to the corresponding plug-in. Second, when generating the platform-specific aspects, the plug-in hooks into the generation process to map the domain-specific details to adequate elements of GPAL aspects. These are then compiled with the GPAL compiler.

5.5.6. POPART. POPART [Dinkelaker et al. 2009] is a framework for embedding DSALs into Groovy [König et al. 2007]. It embeds common aspect-oriented syntax and semantics into Groovy as a library. Developers can build new domain-specific aspect-oriented concepts by using the existing concepts from this library.

To implement a DSAL, a developer needs to implement three components. First, the developer reuses a predefined GPAL join point representation or defines a custom domain-specific join point representation. Second, the developer embeds into Groovy a pointcut language that encodes a set of pointcut designators for that join point representation. Third, the developer embeds into Groovy an advice language that encodes a set of domain-specific actions. Because of the use of the embedding approach no special parser implementation is required for the pointcut and advice languages. The developer only needs to define the operational semantics of the DSAL. To compose DSALs, the developer can use the default aspect-oriented composition semantics or extend, if needed, a framework class to provide logic for resolving composition conflicts. This way, developers can configure resolution for controlling coadvising interactions [Dinkelaker 2011b].

POPART has been used to implement an AspectJ-like join point representation and aspect-oriented language for Groovy. This GPAL has been extended with an embedded version of COOL, the DSAL for caching from Havinga et al. [2008], and a DSAL for data compression [Dinkelaker 2011b], including the resolution of coadvising conflicts between the three languages. In addition, aspect support for domain-specific base languages has been demonstrated for several embedded DSLs [Dinkelaker et al. 2010; Dinkelaker 2011a].

POPART supports implementing DSALs that have a DSL as the base language. Because no DSAL parser has to be implemented from scratch, there are reduced implementation costs. Unfortunately, because the DSAL syntax and semantics is embedded as library calls in Groovy, performance is slow compared to compiler-based implementations.

5.6. Implementation of the Different DSALs

As mentioned before, we have classified the different DSALs according to how they have been implemented. This is shown in Table III. The most notable observation is that almost half of the languages we studied (17 out of 36) use a preprocessor to perform an ad hoc extension to an existing compiler. The absence of entries in the categories "embedding" and "Hybrids using AspectJ" is because we did not find any publications presenting a DSAL that has been implemented using these implementation patterns. Instead, work in these categories has been limited to developing the approach, illustrating it with example languages that we did not include in our survey. Also, despite of a significant body of work on infrastructures being present, we find that only a few languages make use of these infrastructures. Last, we have not been able to classify all studied languages. Two publications state using a compiler, while three more do not provide information on the implementation.

Table III. Classification of the Different DSALs Studied According to Their Implementation Pattern
The names of the languages are as in Tables I and II; the classifications have been detailed in the preceding subsections.

Interpreters	From scratch	*(none)*
	Ad hoc extension	A04BPEL
	Extensible base	Leasing_in_manets
Compilers	From scratch	ScatterML
	Ad hoc extension	AspectLISA, AspectStratego, DiSL
	Ad hoc w. preproc.	Alert, ALPH, AML, AspectG, AspectGrid, COOL, DAJ, DSAL_for_Matlab, ERTSAL, LARA, Padus, PCSL, PDL, RG, RIDL, Robust, VMADL
	Extensible base	AspectMatlab, Conspects
	Unspecified	QuO-ASL, RemoteJ
Embedding		*(none)*
Hybrids	w. Preproc.	AOWP-PHP
	Using AspectJ	*(none)*
Infrastructure		AO4FSM, AO4SQL, KALA, POM, Racer
Unknown		AspectASF, Commentweaver, D4OL

6. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this article, we surveyed the field of DSALs. Based on a corpus of 36 DSALs, we provided a definition of DSALs and discussed their relevant properties. Furthermore, we structured the DSAL design space in the light of the three axes of join point representation, means of identifying at join points, and means of effecting at join points. Last, we also provided a detailed taxonomy of DSAL implementation approaches.

This article is useful for the general software development community first because it discusses a well-defined set of domain-specific languages that may directly be useful to the general practitioner. Second, it guides software developers that wish to implement a new DSAL. Moreover, for DSAL and DSL researchers this work provides a frame of reference for the efforts performed in the research community. This allows existing work to be clearly described in terms of the definitions proposed here, as well as allowing it to be accurately located in the design space.

This survey also highlights areas that have been lacking attention so far and merit further research. More specifically, the following possible research directions warrant attention:

—The text focuses on the three dimensions of join point representation, means of identification, and means of effecting (see Section 3.1). We can, however, also consider a fourth dimension: the domain-specific nature of the base language. The effect of this dimension on the other three has not been studied in detail. We found no case of a DSAL that is DS in the fourth dimension, and GP in the first three. It is not immediately clear whether such a case is possible; further research would be required to establish this (e.g., by constructing such a language).
—In Section 4.1 we have seen that all DSALs are asymmetric: aspect code differs from base code (e.g., one typical difference is that execution of the latter does not emit join points). The reason for this choice is not given by the language designers and it is not clear if a fully symmetric DSAL is possible and whether it would yield any advantages.
—Only few DSALs perform domain-specific optimizations. When we consider the optimization of join point generation, as discussed in Section 4.6, there are, however, clear opportunities. This is illustrated by the three DSALs that perform optimization.

It is not clear why the majority of DSAL implementers have chosen to ignore optimization issues. More research is needed to determine the motivations for this design choice.

—Composition of DSALs is a relevant issue, as there are multiple cases listed of where more than one DSAL is expected to be used in the same codebase. However, as seen in Section 4.7, the only DSALs that provide support for composition are the ones that have been explicitly designed to be composed together. Outside of this space there is no work showing the composition of different DSALs. A possible exception could be the small example languages shown in the work on infrastructures. However, these examples are arguably too small to be representative for full-fledged DSALs and the composition issues that may arise there. A case study on the construction of multiple DSALs using a given infrastructure and an in-depth evaluation of their composability would be valuable.

—Continuing with infrastructures, in Section 5.5, we have seen that only 5 of the 36 DSALs we studied use a DSAL infrastructure for their implementation. The reasons for this low adoption of infrastructures are unknown. This may vary from the worst case: that these do not address the real issues that DSAL developers face, to the case that simply (and sadly) DSAL developers are not aware of their existence. An investigation into the motivation for the nonuse of infrastructure is warranted. This might also reveal why the use of a preprocessor is such a popular implementation technique.

In closing, we hope that our structuring of the design space is useful to programmers and researchers alike, and may serve as a catalyst to spur the further growth of DSALs.

REFERENCES

Michael Achenbach and Klaus Ostermann. 2010a. A meta-aspect protocol for developing dynamic analyses. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer-Verlag, Berlin, 153–167. DOI:http://dx.doi.org/10.1007/978-3-642-16612-9_13

Michael Achenbach and Klaus Ostermann. 2010b. Growing a dynamic aspect language in ruby. In *Proceedings of the 5th Domain-Specific Aspect Languages Workshop at AOSD 2010 (DSAL'10)*.

Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. 2003. *Business Process Execution Language for Web Services Specification*. Technical Report. BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems.

Yoav Apter, David H. Lorenz, and Oren Mishali. 2012. A debug interface for debugging multiple domain specific aspect languages. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development (AOSD'12)*. ACM, New York, NY, 47–58. DOI:http://dx.doi.org/10.1145/2162049.2162056

Toheed Aslam, Jesse Doherty, Anton Dubrau, and Laurie Hendren. 2010. AspectMatlab: An aspect-oriented scientific programming language. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD'10)*. ACM, New York, NY, 181–192.

Ali Assaf and Jacques Noyé. 2008. Dynamic AspectJ. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS'08)*. ACM, New York, NY, Article 8 (2008), 12 pages. DOI:http://dx.doi.org/10.1145/1408681.1408689

Pavel Avgustinov, Aske Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 2006. abc: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development I*, Awais Rashid and Mehmet Aksit (Eds.). Lecture Notes in Computer Science, Vol. 3880. Springer-Verlag, Berlin, 293–334. DOI:http://dx.doi.org/10.1007/11687061_9

Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. 2008. Modularity first: A case for mixing AOP and attribute grammars. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*. ACM, New York, NY, 25–35. DOI:http://dx.doi.org/10.1145/1353482.1353486

Anya Helene Bagge, Valentin David, Magne Haveraaen, and Karl Trygve Kalleberg. 2006. Stayin' alert:: Moulding failure and exceptions to your needs. In *Proceedings of the 5th International Conference*

on Generative Programming and Component Engineering (GPCE'06). ACM, New York, NY, 265–274. DOI:http://dx.doi.org/10.1145/1173706.1173747

Jason Baker and Wilson Hsieh. 2002. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*. ACM, New York, NY, 86–95.

Anindya Basu. 1997. *A Language-Based Approach to Protocol Construction*. Ph.D. dissertation. Cornell University, Ithaca, NY.

Lodewijk Bergmans and Mehmet Aksit. 2001. Composing crosscutting concerns using composition filters. *Commun. ACM* 44, 10 (Oct. 2001), 51–57.

Eric Bodden. 2005. Concern Specific Languages and their implementation with abc. In *Proceedings of the Software-Engineering Properties of Languages and Aspect Technologies workshop (SPLAT!) at the 4th International Conference on Aspect-Oriented Software Development*. (March 2005).

Eric Bodden and Klaus Havelund. 2008. Racer: Effective race detection using AspectJ. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM, New York, NY, 155–166. DOI:http://dx.doi.org/10.1145/1390630.1390650

Mathieu Braem, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, and Viviane Jonckers. 2007. Concern-specific languages in a visual web service creation environment. *Electron. Notes Theor. Comput. Sci.* 163, 2 (April 2007), 3–17. DOI:http://dx.doi.org/DOI: 10.1016/j.entcs.2006.10.012

Mathieu Braem, Kris Verlaenen, Niels Joncheere, Wim Vanderperren, Ragnhild Straeten, Eddy Truyen, Wouter Joosen, and Viviane Jonckers. 2006. Isolating process-level concerns using padus. In *Business Process Management*, Schahram Dustdar, JosLuiz Fiadeiro, and AmitP. Sheth (Eds.). Lecture Notes in Computer Science, Vol. 4102. Springer-Verlag, Berlin, 113–128. DOI:http://dx.doi.org/10.1007/11841760_9

M. Bravenboer and E. Visser. 2004. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM , New York, NY, 365–383.

David Bruce. 1997. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In *DSL'97—First ACM SIGPLAN Workshop on Domain-Specific Languages, in Association with POPL'97*. 17–35. University of Illinois Computer Science Report.

M. Bruntink, A. van Deursen, and T. Tourwe. 2005. Isolating idiomatic crosscutting concerns. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, Los Alamitos, CA, 37–46. DOI:http://dx.doi.org/10.1109/ICSM.2005.57

Joao Cardoso, Pedro Diniz, Miguel P. Monteiro, João M. Fernandes, and João Saraiva. 2010. A domain-specific aspect language for transforming MATLAB programs. In *Proceedings of the 5th Domain-Specific Aspect Languages Workshop at AOSD 2010 (DSAL'10)*.

João M. P. Cardoso, Tiago Carvalho, José G. F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz, and Zlatko Petrov. 2012. LARA: An aspect-oriented programming language for embedded systems. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development (AOSD'12)*. ACM, New York, NY, 179–190. DOI:http://dx.doi.org/10.1145/2162049.2162071

Denis Caromel, Luis Mateu, Guillaume Pothier, and Éric Tanter. 2008. Parallel object monitors. *Concurrency Comput.: Pract. Exper.* 20, 12 (Aug. 2008), 1387–1417.

Denis Caromel, Luis Mateu, and Éric Tanter. 2004. Sequential object monitors. In *ECOOP 2004—Object-Oriented Programming*, Martin Odersky (Ed.). Lecture Notes in Computer Science, Vol. 3086. Springer-Verlag, Berlin, 316–340.

Andrew Casey, Jun Li, Jesse Doherty, Maxime Chevalier-Boisvert, Toheed Aslam, Anton Dubrau, Nurudeen Lameed, Amina Aslam, Rahul Garg, Soroush Radpour, Olivier Savary Belanger, Laurie Hendren, and Clark Verbrugge. 2010. McLab: An extensible compiler toolkit for MATLAB and related languages. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering (C3S2E'10)*. ACM, New York, NY, 114–117. DOI:http://dx.doi.org/10.1145/1822327.1822343

Anis Charfi and Mira Mezini. 2007. AO4BPEL: An aspect-oriented extension to BPEL. *World Wide Web* 10, 3 (Sept. 2007), 309–344. DOI:http://dx.doi.org/10.1007/s11280-006-0016-3

Panos K. Chrysanthis and Krithi Ramamritham. 1991. A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB'91)*. Morgan Kaufmann, Burlington, MA, 103–112.

Thomas Cleenewerck, Jacques Noyé, Johan Fabry, Anne-Françoise Le Meur, and Éric Tanter. 2008. Summary of the third workshop on domain-specific aspect languages. In *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages (DSAL'08)*. ACM, New York, NY, 1–5.

Pierre-Charles David, Thomas Ledoux, and Noury M. Bouraqadi-Sadani. 2001. Two-step weaving with reflection using AspectJ. In *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. Retrieved from http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/02-david.pdf.

Kris De Volder and Theo D'Hondt. 1999. Aspect-oriented logic meta programming. In *Meta-Level Architectures and Reflection*, Pierre Cointe (Ed.). Lecture Notes in Computer Science, Vol. 1616. Springer-Verlag, Berlin, 250–272. DOI:http://dx.doi.org/10.1007/3-540-48443-4_22

Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo DHondt, and Wolfgang De Meuter. 2006. Ambient-oriented programming in AmbientTalk. In *ECOOP 2006—Object-Oriented Programming*, Dave Thomas (Ed.). Lecture Notes in Computer Science, Vol. 4067. Springer-Verlag, Berlin, 230–254. DOI:http://dx.doi.org/10.1007/11785477_16

Edsger W. Dijkstra. 1982. *Selected writings on Computing: A Personal Perspective*. Springer-Verlag, New York, NY, 60–66.

Tom Dinkelaker. 2011a. AO4SQL: Towards an aspect-oriented extension for SQL. In *Proceedings of the 8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'11)*.

Tom Dinkelaker. 2011b. *Reflective Embedding of Domain-Specific Languages*. Ph.D. dissertation. Technische Universität Darmstadt, Darmstadt, Germany.

Tom Dinkelaker, Michael Eichberg, and Mira Mezini. 2010. An architecture for composing embedded domain-specific languages. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD'10)*. ACM, New York, NY, 49–60. DOI:http://dx.doi.org/10.1145/1739230.1739237

Tom Dinkelaker, Michael Eichberg, and Mira Mezini. 2013. Incremental concrete syntax for embedded languages with support for separate compilation. *Science of Computer Programming* 78, 6 (June 2013), 615–632. DOI:http://dx.doi.org/10.1016/j.scico.2012.12.002

Tom Dinkelaker, Mohammed Erradi, and Meryeme Ayache. 2012. Using aspect-oriented state machines for detecting and resolving feature interactions. *Comput. Sci. Inf. Syst.* 9, 3 (2012), 1045–1074.

Tom Dinkelaker and Mira Mezini. 2008. Dynamically linked domain-specific extensions for advice languages. In *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages (DSAL'08)*. ACM, New York, NY, Article 3, (2008) 7 pages. DOI:http://dx.doi.org/10.1145/1404927.1404930

Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. 2009. The art of the meta-aspect protocol. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD'09)*. ACM, New York, NY, 51–62.

Jesse Doherty. 2010. (Oct. 2010). Personal communication.

Rémi Douence, Pascal Fradet, and Mario Südholt. 2002. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*. Springer-Verlag, Berlin, 173–188.

Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. 2006. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Prog.* 63, 3 (Dec. 2006), 207–239.

Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. 2004. Building adaptive distributed applications with middleware and aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*. ACM, New York, NY, 66–73. DOI:http://dx.doi.org/10.1145/976270.976280

Torbjörn Ekman and Görel Hedin. 2007. The JastAdd extensible Java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. ACM, New York, NY, 1–18. DOI:http://dx.doi.org/10.1145/1297027.1297029

Patrick Eugster and K.R. Jayaram. 2009. EventJava: An extension of java for event correlation. In *ECOOP 2009—Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Lecture Notes in Computer Science, Vol. 5653. Springer-Verlag, Berlin, 570–594. DOI:http://dx.doi.org/10.1007/978-3-642-03013-0_26

Johan Fabry, Éric Tanter, and Theo D'Hondt. 2008. KALA: Kernel aspect language for advanced transactions. *Sci. Comput. Prog.* 71, 3 (May 2008), 165–180.

J. Fabry, E. Tanter, and T. D'Hondt. 2009. Infrastructure for domain-specific aspect languages: The ReLAx case study. *IET Software* 3, 3 (June 2009), 238–254. DOI:http://dx.doi.org/10.1049/iet-sen.2007.0120

Marc Fleury and Francisco Reverbel. 2003. The JBoss extensible server. In *Middleware 2003*, Markus Endler and Douglas Schmidt (Eds.). Lecture Notes in Computer Science, Vol. 2672. Springer-Verlag, Berlin, 344–373. DOI:http://dx.doi.org/10.1007/3-540-44892-6_18

Martin Fowler. 2010. *Domain-Specific Languages*. Addison-Wesley Professional, Boston, MA.

Pascal Fradet and Mario Südholt. 1999. An aspect language for robust programming. In *Proceedings of the International Workshop on Aspect-Oriented Programming at ECOOP'99*.

Debasish Ghosh. 2010. *DSLs in Action*. Manning, Shelter Island, NY.

Elisa Gonzalez Boix, Thomas Cleenewerk, Jessie Dedecker, and Wolfgang De Meuter. 2008. Towards a domain-specific aspect language for leasing in mobile ad hoc networks. In *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages (DSAL'08)*. ACM, New York, NY, Article 6, 5 pages. DOI:http://dx.doi.org/10.1145/1404927.1404933

Jan Hannemann and Gregor Kiczales. 2002. Design pattern implementation in Java and AspectJ. *SIGPLAN Not.* 37, 11 (Nov. 2002), 161–173. DOI:http://dx.doi.org/10.1145/583854.582436

William Harrison, Harold Ossher, and Peri Tarr. 2002. *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*. Technical Report RC22685. IBM Research Division.

M. Haupt, C. Gibbs, B. Adams, S. Timbermont, Y. Coady, and R. Hirschfeld. 2009. Disentangling virtual machine architecture. *IET Software,* 3, 3 (June 2009), 201–218. DOI:http://dx.doi.org/10.1049/iet-sen.2007.0121

Wilke Havinga, Lodewijk Bergmans, and Mehmet Aksit. 2008. Prototyping and composing aspect languages. In *ECOOP 2008—Object-Oriented Programming*, Jan Vitek (Ed.), Lecture Notes in Computer Science, Vol. 5142. Springer-Verlag, Berlin, 180–206. DOI:http://dx.doi.org/10.1007/978-3-540-70592-5_9

Jan Heering and Marjan Mernik. 2007. *Domain-Specific Languages in Perspective*. Technical Report SEN-E0702. CWI, Amsterdam, The Netherlands. Retrieved from from http://oai.cwi.nl/oai/asset/12319/12319D.pdf.

Erik Hilsdale and Jim Hugunin. 2004. Advice weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*. ACM, New York, NY, 26–35. DOI:http://dx.doi.org/10.1145/976270.976276

Robert Hirschfeld. 2003. AspectS—Aspect-oriented programming with squeak. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, Mehmet Aksit, Mira Mezini, and Rainer Unland (Eds.). Lecture Notes in Computer Science, Vol. 2591. Springer-Verlag, Berlin, 216–232. DOI:http://dx.doi.org/10.1007/3-540-36557-5_17

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*. ACM Press, New York, NY, 137–148.

Keiji Hokamura, Naoyasu Ubayashi, Shin Nakajima, and Akihito Iwai. 2008. Aspect-oriented programming for web controller layer. *Asia-Pacific Software Engineering Conference* 0 (2008), 529–536. DOI:http://dx.doi.org/10.1109/APSEC.2008.69

Adrian Holzer, Lukasz Ziarek, K.R. Jayaram, and Patrick Eugster. 2011. Putting events in context: Aspects for event-based distributed programming. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11)*. ACM Press, New York, NY, 241–252. DOI:http://dx.doi.org/10.1145/1960275.1960304

Michihiro Horie and Shigeru Chiba. 2010. Tool support for crosscutting concerns of API documentation. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD'10)*. ACM Press, New York, NY, 97–108. DOI:http://dx.doi.org/10.1145/1739230.1739242

Paul Hudak. 1996. Building domain-specific embedded languages. *Comput. Surveys* 28, 4es (Dec. 1996).

John Irwin, Jean-Marc Loingtier, JohnR. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. 1997. Aspect-oriented programming of sparse matrix code. In *Scientific Computing in Object-Oriented Parallel Environments*, Yutaka Ishikawa, Rodney R. Oldehoeft, John V. W. Reynders, and Marydell Tholburn (Eds.). Lecture Notes in Computer Science, Vol. 1343. Springer-Verlag, Berlin, Heidelberg, 249–256. DOI:http://dx.doi.org/10.1007/3-540-63827-X_68

Karl Trygve Kalleberg and Eelco Visser. 2006. Combining aspect-oriented and strategic programming. *Electronic Notes Theor. Comput. Sci.* 147, 1 (January 2006), 5–30. DOI:http://dx.doi.org/10.1016/j.entcs.2005.06.035

G. Kiczales, J. Des Rivieres, and D.G. Bobrow. 1991. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, USA.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and WilliamG. Griswold. 2001. An overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Lecture Notes in Computer Science, Vol. 2072. Springer-Verlag, Berlin, 327–354. DOI:http://dx.doi.org/10.1007/3-540-45337-7_18

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming*, Mehmet Aksit and Satoshi Matsuoka (Eds.). Lecture Notes in Computer Science, Vol. 1241. Springer-Verlag, Berlin, 220–242. DOI:http://dx.doi.org/10.1007/BFb0053381

Paul Klint, Tijs Storm, and Jurgen Vinju. 2005. Term rewriting meets aspect-oriented programming. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, Aart Middeldorp, Vincent Oostrom, Femke

Raamsdonk, and Roel Vrijer (Eds.). Lecture Notes in Computer Science, Vol. 3838. Springer-Verlag, Berlin, 88–105. DOI: http://dx.doi.org/10.1007/11601548_8

Sergei Kojarski and David H. Lorenz. 2004. AOP as a first class reflective mechanism. In *OOPSLA Companion (OOPSLA'04)*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, New York, NY, 216–217. DOI: http://dx.doi.org/10.1145/1028664.1028757

Sergei Kojarski and David H. Lorenz. 2005. Pluggable AOP: Designing aspect mechanisms for third-party composition. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM, New York, NY, 247–263. DOI: http://dx.doi.org/10.1145/1094811.1094831

Sergei Kojarski and David H. Lorenz. 2007. 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 and Applications (OOPSLA'07)*. ACM, New York, NY, 515–534. DOI: http://dx.doi.org/10.1145/1297027.1297065

Dierk König, Aandrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. 2007. *Groovy in Action*. Manning, Shelter Island, NY.

T. Kosar, M. López, E. Pablo, P.A. Barrientos, and M. Mernik. 2008. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.* 50, 5 (April 2008), 390–405.

Toma Kosar, Marjan Mernik, and Jeffrey C. Carver. 2012. Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments. *Empirical Softw. Eng.* 17, 3 (2012), 276–304. DOI: http://dx.doi.org/10.1007/s10664-011-9172-x

Tomaž Kosar, Marjan Mernik, João Maria Pereira, Varanda, Matej Črepinšek, Daniela Da Cruz, and Rangel Pedro Henriques. 2010. Comparing general-purpose and domain-specific languages: An empirical study. *Comput. Sci. Inf. Syst.* 7, 2 (2010), 247–264.

Cristina Videira Lopes and Trung Chi Ngo. 2004. *The Aspect Markup Language and Its Support of Aspect Plugins*. ISR Technical Report UCI-ISR-04-8. University of California, Irvine.

Cristina Videira Lopes. 1997. *D: A Language Framework for Distributed Programming*. Ph.D. dissertation. College of Computer Science of Northeastern University, Boston, MA.

David Lorenz and Sergei Kojarski. 2007. Understanding aspect interactions, co-advising and foreign advising. In *Proceedings of the Second International Workshop on Aspects, Dependencies and Interactions at ECOOP07*, Frans Sanen, Ruzanna Chitchyan, Lodewijk Bergmans, Johan Fabry, and Mario Südholt (Eds.). Technical Report CW 497. Katholieke Universiteit Leuven, Leuven, 23–28.

David H. Lorenz. 2012. Language-oriented modularity through awesome DSALs: Summary of invited talk. In *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages (DSAL'12)*. ACM, New York, NY, 1–2. DOI: http://dx.doi.org/10.1145/2162037.2162039

David H. Lorenz and Oren Mishali. 2012. SPECTACKLE: Toward a specification-based DSAL composition process. In *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages (DSAL'12)*. ACM, New York, NY, 9–14. DOI: http://dx.doi.org/10.1145/2162037.2162042

Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A domain-specific language for bytecode instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development (AOSD'12)*. ACM, New York, NY, 239–250. DOI: http://dx.doi.org/10.1145/2162049.2162077

Antoine Marot and Roel Wuyts. 2008. A DSL to declare aspect execution order. In *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages (DSAL'08)*. ACM, New York, NY, Article 7, 5 pages. DOI: http://dx.doi.org/10.1145/1404927.1404934

Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. 2006. A fine-grained join point model for more reusable aspects. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Lecture Notes in Computer Science, Vol. 4279. Springer-Verlag, Berlin, 131–147. DOI: http://dx.doi.org/10.1007/11924661_8

Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. 2003. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction. CC'03*, Grel Hedin (Ed.). Lecture Notes in Computer Science, Vol. 2622. Springer-Verlag, Berlin, Heidelberg, 46–60. DOI: http://dx.doi.org/10.1007/3-540-36579-6_4

Anurag Mendhekar, Gregor Kiczales, and John Lamping. 1997. *RG: A Case-Study for Aspect-Oriented Programming*. Technical Report SPL97-009. XEROX PARC.

Marjan Mernik (Ed.). 2013. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, Hershey, PA.

Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *Comput. Surveys* 37, 4 (Dec. 2005), 316–344. DOI: http://dx.doi.org/10.1145/1118890.1118892

Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. 2002. LISA: An interactive environment for programming language development. In *Compiler Construction (CC'02)*, R. Nigel Horspool (Ed.). Lecture Notes in Computer Science, Vol. 2304. Springer-Verlag, Berlin, 1–4. DOI:http://dx.doi.org/10.1007/3-540-45937-5_1

Clint Morgan, Kris De Volder, and Eric Wohlstadter. 2007. A static aspect language for checking design rules. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07)*. ACM, New York, NY, 63–72. DOI:http://dx.doi.org/10.1145/1218563.1218571

J. Munnelly and S. Clarke. 2008. A domain-specific language for ubiquitous healthcare. In *Third International Conference on Pervasive Computing and Applications, 2008 (ICPCA'08)*, Vol. 2, 757–762. DOI:http://dx.doi.org/10.1109/ICPCA.2008.4783710

J. Noble, A. Schmidmeier, D. J. Pearce, and A. P. Black. 2007. Patterns of aspect-oriented design. In *Proceedings of European Conference on Pattern Languages of Program (EuroPLoP'07)*, Lise B. Hvatum and Till Schümmer (Eds.). UVK–Universitaetsverlag Konstanz, Irsee, Germany.

Amjad Nusayr and Jonathan Cook. 2009. AOP for the domain of runtime monitoring: Breaking out of the code-based model. In *Proceedings of the 4th Workshop on Domain-Specific Aspect Languages (DSAL'09)*. ACM, New York, NY, 7–10. DOI:http://dx.doi.org/10.1145/1509307.1509310

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: An extensible compiler framework for java. In *Compiler Construction (CC'03)*, Görel Hedin (Ed.). Lecture Notes in Computer Science, Vol. 2622. Springer-Verlag, Berlin, 138–152. DOI:http://dx.doi.org/10.1007/3-540-36579-6_11

Harold Ossher and Peri Tarr. 2002. Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology*, Mehmet Aksit (Ed.). The Springer International Series in Engineering and Computer Science, Vol. 648. Springer, New York, NY, 293–323. DOI:http://dx.doi.org/10.1007/978-1-4615-0883-0_10

David L. Parnas. 1972. On the criteria for decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058.

T. J. Parr and R. W. Quong. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience* 25, 7 (July 1995), 789–810. DOI:http://dx.doi.org/10.1002/spe.4380250705

D. Rebernak, M. Mernik, H. Wu, and J. Gray. 2009. Domain-specific aspect languages for modularising crosscutting concerns in grammars. *IET Software* 3, 3 (June 2009), 184–200. DOI:http://dx.doi.org/10.1049/iet-sen.2007.0114

Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. 2010. Embedding languages without breaking tools. In *ECOOP 2010 Object-Oriented Programming*, Theo D'Hondt (Ed.). Lecture Notes in Computer Science, Vol. 6183. Springer-Verlag, Berlin, 380–404. DOI:http://dx.doi.org/10.1007/978-3-642-14107-2_19

Walter A. Rolling. 1994. A preliminary annotated bibliography on domain engineering. *SIGSOFT Softw. Eng. Notes* 19, 3 (July 1994), 82–84. DOI:http://dx.doi.org/10.1145/182824.182844

J. L. Sobral, M. P. Monteiro, and C. A. Cunha. 2006. Aspect-oriented support for modular parallel computing. In *Proceedings of the 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. 37–41.

João L. Sobral and Miguel P. Monteiro. 2008. A domain-specific language for parallel and grid computing. In *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages (DSAL'08)*. ACM Press, New York, NY, Article 2, 4 pages. DOI:http://dx.doi.org/10.1145/1404927.1404929

Paul Soule. 2010. *Autonomics Development: A Domain-Specific Aspect Language Approach*. Springer Basel, Basel. DOI:http://dx.doi.org/10.1007/978-3-0346-0540-3_3

Paul Soule, Tom Carnduff, and Stuart Lewis. 2007. A distribution definition language for the automated distribution of Java objects. In *Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (DSAL'07)*. ACM Press, New York, NY. DOI:http://dx.doi.org/10.1145/1255400.1255402

William Sousan, Victor Winter, Mansour Zand, and Harvey Siy. 2007. ERTSAL: A prototype of a domain-specific aspect language for analysis of embedded real-time systems. In *Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (DSAL'07)*. ACM, New York, NY, Article 1. DOI:http://dx.doi.org/10.1145/1255400.1255401

Johan J Sung and Karl Lieberherr. 2002. *DAJ: A Case Study of Extending AspectJ*. Technical Report NU-CCS-02-16. Northeastern University.

Éric Tanter. 2004. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. Ph.D. dissertation. University of Nantes and University of Chile, Nantes, France and Santiago, Chile.

Éric Tanter. 2006. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, Welf Löwe and Mario Südholt (Eds.). Lecture Notes in Computer Science, Vol. 4089. Springer-Verlag, Berlin, 98–113.

Éric Tanter and Johan Fabry. 2009. Supporting composition of structural aspects in an AOP kernel. *J. Univ. Comput. Sci.* 15, 3 (2009), 620–647. DOI:http://dx.doi.org/10.3217/jucs-015-03-0620

Éric Tanter and Jacques Noyé. 2005. A versatile kernel for multi-language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'05)*, Robert Glück and Mike Lowry (Eds.). Lecture Notes in Computer Science, Vol. 3676. Springer-Verlag, Berlin, 173–188.

Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. 2003. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, Ron Crocker and Guy L. Steele, Jr. (Eds.). ACM, New York, NY, 27–46.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. ACM, New York, NY, 107–119.

Stijn Timbermont, Bram Adams, and Michael Haupt. 2008. Towards a DSAL for object layout in virtual machines. In *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages (DSAL'08)*. ACM, New York, NY, Article 5, 4 pages. DOI:http://dx.doi.org/10.1145/1404927.1404932

Rodolfo Toledo and Éric Tanter. 2008. A lightweight and extensible AspectJ implementation. *J. Univ. Comput. Sci.* 14, 21 (2008), 3517–3533.

Laurence Tratt. 2008. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.* 30, 6 (Oct. 2008), 31:1–31:40. DOI:http://dx.doi.org/10.1145/1391956.1391958

Naoyasu Ubayashi, Hidehiko Masuhara, and Tetsuo Tamai. 2004. An AOP implementation framework for extending join point models. In *Proceedings of the ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04)*, Walter Cazzola, Shigeru Chiba, and Gunter Saake (Eds.). Oslo, Norway.

M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. 2001. The ASF+SDF meta-environment: A component-based language development environment. In *Compiler Construction (CC'01)*, Reinhard Wilhelm (Ed.). Lecture Notes in Computer Science, Vol. 2027. Springer-Verlag, Berlin, 365–370. DOI:http://dx.doi.org/10.1007/3-540-45306-7_26

Arie van Deursen and Paul Klint. 1998. Little languages: Little maintenance? *J. Softw. Maint.: Res. Pract.* 10, 2 (1998), 75–92.

Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.* 35, 6 (June 2000), 26–36. DOI:http://dx.doi.org/10.1145/352029.352035

Eelco Visser. 2004. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, Lecture Notes in Computer Science, Vol. 3016. Springer-Verlag, Berlin, 216–238.

C. Walls and R. Breidenbach. 2005. *Spring in Action*. Manning, Shelter Island, NY.

D. Wampler. 2008. Aquarium: AOP in Ruby. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development—Industry Track (AOSD'07)*. 60–67.

Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. 2004. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Prog. Lang. Syst.* 26, 5 (Sept. 2004), 890–910.

J. White and D. C. Schmidt. 2009. Automating deployment planning with an aspect weaver. *IET Software,* 3, 3 (June 2009), 167–183. DOI:http://dx.doi.org/10.1049/iet-sen.2007.0123

Received March 2013; revised July 2014; accepted October 2014