On Dynamically-Scoped Crosscutting Mechanisms

Éric Tanter
DCC – University of Chile
http://www.dcc.uchile.cl/˜etanter

Many crosscutting mechanisms proposed in the literature offer means to restrain aspects to some dynamically-defined scopes. Dynamically-scoped mechanisms are particularly interesting because of the flexibility, expressiveness, and control they give over structural and behavioral changes made by aspects. Since the nature of dynamic scopes as well as the scoping mechanisms themselves greatly vary among proposals, it is relatively complex to compare them. This paper aims at filling this gap by proposing a first characterization of dynamically-scoped crosscutting mechanisms, hence providing a reference frame for comparing different approaches. As a result, this work clarifies some differences between related approaches, suggests possible tracks for further exploration of the design space of such mechanisms, and discusses some issues raised by dynamically-scoped aspects.

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

General Terms: Languages

Additional Key Words and Phrases: Aspect-oriented programming, crosscutting mechanisms, dynamic scope, aspects

1. INTRODUCTION

Aspect-Oriented Programming [Elrad et al. 2001] and other related approaches propose mechanisms for the modularization of crosscutting concerns. Although all these mechanisms were initially statically-scoped, there is a growing trend of interest in dynamic scoping mechanisms in the field. This interest in dynamically-scoped crosscutting mechanisms is justified by the flexibility and expressiveness offered by such mechanisms, that make it possible to conveniently express structural and behavioral changes that should only occur under certain dynamic circumstances.

It is important to note that the notion of dynamic scope we use in this paper slightly differs from the traditional one: a dynamically-scoped crosscutting mechanism is not only a mechanism that scopes an aspect according to thread-local values or code, but also to possibly any past, current, and surrounding states of the program. Examples include conditionals like `if` pointcuts in AspectJ and others, control and data flow conditions, tracematches and stateful aspects, and context-aware aspects [Kiczales et al. 2001; Hirschfeld 2002; Masuhara and Kawauchi 2003; Douence et al. 2005; Allan et al. 2005; Douence et al. 2004; Vanderperren et al. 2005; Aracic et al. 2006; Tanter et al. 2006]. Similarly, related proposals which are not AOP as such, but still promote better modularization of crosscutting concerns, such as metaobject protocols and structural refinements, have introduced dynamic scoping mechanisms [Tanter et al. 2003; Bergel et al. 2003; Costanza and Hirschfeld 2005].

As a side effect of this proliferation of dynamic scoping mechanisms, it is relatively complex to compare each proposal with respect to how dynamic scoping is actually provided. In this paper we propose a characterization of dynamically-scoped crosscutting mechanisms. The objective of this classification is to give a reference frame for comparing different approaches. Therefore, beyond the classification itself, this paper contributes to the field by *(a)* clarifying the differences between related approaches, *(b)* highlighting similar combination of characteristics of some approaches, thus calling for deeper comparison, and *(c)* pointing out some tracks for further exploring the design space of dynamically-scoped crosscutting mechanisms.

In the following, we use aspect in a broad sense, which includes mechanisms for handling crosscutting such as Classboxes [Bergel et al. 2003] and mixin layers as provided in ContextL [Costanza and Hirschfeld 2005].

2. CHARACTERIZING DYNAMIC SCOPE

An aspect definition includes the (usually intentional) definition of program or execution points of interest (*a.k.a.* the cut of the aspect) over which some action is to be performed. The cut of an aspect has a static part, and possibly a dynamic part –in the case of a dynamically-scoped aspect–. The action of the aspect can either modify the structure or behavior of a program. We refer to these as structural and behavioral aspects, respectively. For instance,

Author's address: É. Tanter, Computer Science Department, University of Chile, Blanco Encalada 2120, Santiago, Chile.
A previous version of this work was presented at the European Workshop on Aspects in Software (EWAS 2006), held in Twente, The Netherlands, September 2006.

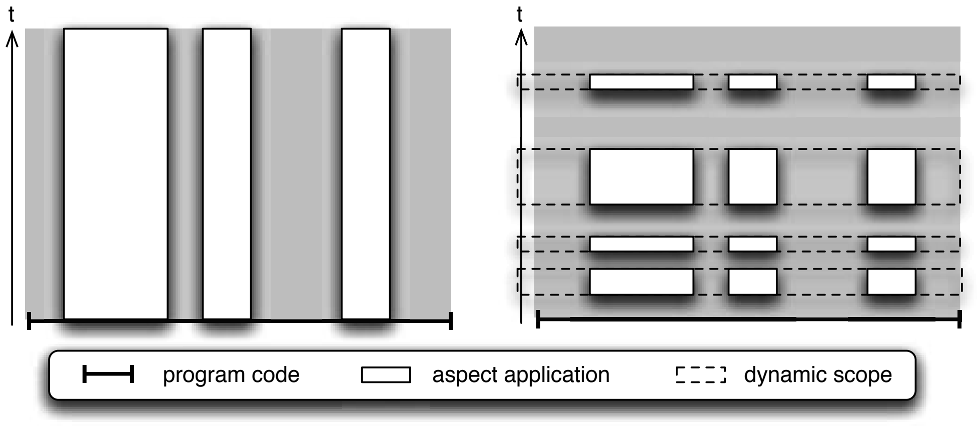

Fig. 1. General view of the application of a statically-scoped aspect (left) and a dynamically-scoped aspect (right).

a monitoring aspect modifies the behavior of an application in order to report on some parts of the application execution; a caching aspect adds a cache field to certain classes, thereby modifying their structure.

2.1 Dynamically-Scoped Aspects in a Nutshell

The application of an aspect is always statically scoped: the cut of an aspect includes some criteria that relate to the parts of the program text that are potentially altered by the aspect. Let us represent the entire code of a program as a uni-dimensional line. If we consider time, that is, program execution, then the application of a statically-scoped aspect is depicted in Fig. 1(left): the parts of the program that are affected by the aspect are *always* affected, from the start to the end of the execution.

A dynamically-scoped aspect is different. Depending on the program being in a certain scope or not during execution, then the aspect applies or not. This is depicted in Fig. 1(right), where we see that the aspect only applies within a certain dynamic scope, which is active four times in the picture. Note that we indifferently refer to *"the program is in a given scope"* and *"the given scope is active"*. The above relates to an aspect that is subject *as a whole* to a certain scope. Most aspect languages and tools allow for the definition of aspects composed of different subparts, each potentially subject to different dynamic scopes.

Dynamically-scoped aspects are typically implemented using residues: evaluated at runtime, a residue is a condition that determines whether the aspect applies or not [Masuhara et al. 2003; Hilsdale and Hugunin 2004]. Dynamically-scoped structural changes are usually performed once for all, and an additional logic ensures that the changes are visible only in the desired dynamic scope.

2.2 Dimensions of Characterization

We propose several dimensions of characterization for dynamically-scoped crosscutting mechanisms. These dimensions have been chosen by focusing on semantics and software engineering issues. We do not consider the mechanisms from a point of view of implementation techniques and possible runtime efficiency. The reason for this is the need we see for a clear *conceptual* framework relating to dynamically-scoped aspects. We identify the following dimensions:

aspect action. What is the nature of the aspect action? It can be either structural (*e.g.* a refinement as in Classboxes [Bergel et al. 2003] and mixin layers [Costanza and Hirschfeld 2005] or an inter-type declaration as in AspectJ) or behavioral (*a.k.a.* advice). The rule to discriminate between a structural change and a behavioral change is that a structural change consists of an aspect adding or explicitly modifying an existing structural element, while a behavioral aspect defines some action to be triggered upon some execution events (this mechanism may be implemented by modifying the structure of the program [Hilsdale and Hugunin 2004], but does so in a manner that is oblivious to the programmer).

scope definition. How are the boundaries (*i.e.* the entry and exit points) of the scope specified by the programmer? Is it done implicitly or explicitly at the points where the scope starts and ends? This characteristic determines the *intrusiveness* of the scope definition (*i.e.* how much base code has to be modified in order to define the scope), as

scoping mechanism	*example systems*	*scope*
if/restriction	*most*	condition true
receiver, target, etc.	*most*	JP dynamic property
1st-class activation	Reflex, AspectS	condition true
cflow	*most*	dyn-ext. of nested pointcut
tracematches	AJ+TM	execution history
context-aware aspects	Reflex extension	any context
deploy block	CaesarJ, Steamloom	dyn-ext. of `deploy`
on/off	AspectS, Reflex	between `on` and `off`
classboxes	Classbox/J	dyn-ext. of top client package
mixin layers	ContextL	dyn-ext. of layer activation

Fig. 2. List of considered dynamic scoping mechanisms.

well as its tractability by the user (*i.e.* how easy it is for the programmer to infer when a given scope is active or not). It has to be noted that in case of intrusive scope definitions, aspects can normally be used to insert start and end points transparently.
Another question, which is highly dependent on the implementation platform (and hence not considered further) is how the scope boundaries are determined by the runtime infrastructure[1].

scope information exposure. Can information associated with the scope (*e.g.* a value of a variable bound in the dynamic scope) be exposed to the aspect action? This characteristic is particulary important to foster reuse of aspect actions, through genericity. Without such a feature, workarounds have to be devised in order to share scope-specific information with the aspect action, if at all possible.

scope-aspect binding. When is the binding between a scope and an aspect made? We use the characterization of binding *time* and *mode* introduced in [Redmond and Cahill 2002]. The *binding time* is the moment at which the binding is established; it can be either compile time[2] or runtime. The *binding mode* is another dimension referring to whether the binding can be undone/redone during execution. If yes, then the binding mode is said to be dynamic, otherwise it is said to be static.

thread locality. Considering a multi-threaded execution, is the scope defined locally for each thread? If not, should thread locality be manually implemented? This characteristic is very important for the actual semantics of the dynamic scoping mechanism, and the programmer should therefore be well aware of the default semantics of the mechanism (*e.g.* a cflow in AspectJ is by default thread local, while a tracematch in the AspectJ extension of [Allan et al. 2005] is by default global). Note that thread locality is explicitly considered in this classification because dynamically-scoped mechanisms are not necessarily related to execution stack properties; they can for instance relate to past execution states, or to elements pertaining to the physical environment of the running system.

3. DYNAMIC SCOPING MECHANISMS

We now analyze a number of proposals for dynamically-scoped aspects, in the light of the different criteria mentioned above. We consider the following proposals: AspectJ [Kiczales et al. 2001] and similar systems, AspectJ with tracematching [Allan et al. 2005], Reflex [Tanter et al. 2003; Tanter and Noyé 2005], AspectS [Hirschfeld 2002], context-aware aspects [Tanter et al. 2006], CaesarJ [Aracic et al. 2006], Classbox/J [Bergel et al. 2003; Bergel et al. 2005], and ContextL [Costanza and Hirschfeld 2005]. The two last approaches are the only dynamically-scoped approaches to structural changes we know of; intertype declarations in AspectJ are statically scoped, and dynamically-scoped structural links in Reflex have only recently been implemented in order to support similar mechanisms. We also only focus on language-level approaches, not architecture-level ones. The different mechanisms are listed and described on Fig. 2, and our analysis is summarized on Fig. 3.

[1]This refers to either "push"-like mode whereby the aspect is effectively installed upon entering the scope, or "pull"-like mode in which the aspect is always installed and a systematic check is performed before evaluating the aspect action. A system which truly supports dynamic weaving can choose the "push" option, while other systems have to resort to "pull" techniques.
[2]For simplicity, we refer to compile, deploy and link times as compile time.

If/restriction. This is the basic means to specify an arbitrary boolean condition to be evaluated at runtime, as with the `if` pointcut in AspectJ or restrictions in Reflex to name a few. This kind of conditions is statically bound at compile time. The dynamic scope is defined by the condition evaluating to true. Such conditions typically cannot expose context information to the aspect action. Thread locality has to be manually handled.

Dynamic join point properties. We hereby refer to dynamic conditions over join point properties, such as receiver or target runtime type, actual arguments, etc. This mechanism is supported by most AOP systems, and usually permits to expose dynamic information to the aspect action. This contrasts with the previous mechanism.

First-class activation. Some AOP systems like AspectS and Reflex support first-class activation conditions. These are objects encapsulating a dynamic condition. The difference with the if/restriction mechanism is binding mode: activation conditions can be unbound/rebound at runtime. So both Reflex and AspectS offer the same binding mode (dynamic), but differ in the binding time due to implementation environments (Java vs. Smalltalk); Reflex offers dynamic binding at compile time, while AspectS has dynamic binding at runtime. First-class activation is typically global to all threads, but thread locality can be developed in a reusable and generic manner [Hirschfeld and Costanza 2005] (hence the `M/A` characterization on Fig. 3).

Cflow. Most AOP languages and systems support the definition of scope based on control flow, like the `cflow` and `cflowbelow` pointcut designators in AspectJ. These are higher-order constructs, and the scope they define is the dynamic extent of the nested pointcut. Although bound statically in AspectJ, in Reflex one can actually choose between a statically-bound restriction or a dynamically-bound activation condition to define a scope based on control flow. An AspectS implementation could actually be dynamically-bound at runtime. A major use of this mechanism is to make information up in the stack available to an aspect action (like in the wormhole design pattern [Laddad 2003]). All control flow implementations are indeed thread local, since control flow is hereby a stack-related property (and each thread has its own stack).

Tracematches. Stateful aspects or tracematches make it possible to restrain the application of an aspect to the occurrences of certain execution event patterns. Compared to the above mechanism, tracematches greatly enhance the expressiveness of the scoping, although not differing significantly in most other regards. An exception that we already mentioned is the thread locality of the tracematch: the default assumption of the AspectJ extension proposed in [Allan et al. 2005] is that the tracematch is checked globally, that is, matching events in all threads of the system. Still, a special `threadlocal` keyword is supported, to easily switch to a thread-local semantics, whereby sequences of events are looked for in the execution history of a single thread. We consider the syntactical equivalent of a reusable activation condition for thread locality. The most mature implementations to date [Vanderperren et al. 2005; Allan et al. 2005] are statically-bound at compile time.

Context-aware aspects. A Reflex framework for context-aware aspects was proposed in [Tanter et al. 2006]. The idea is to extend pointcut languages with context-specific restrictions, allowing both parameterization of context definitions and exposure of context state to the aspect action. Context here refers to more than just the program execution context, as it includes external context perceived by sensors, for instance. The clear separation of aspects and contexts fosters evolution and reuse of both. The dynamic scope definition in this approach is actually the context definition. In addition to being active or not, a context exposes state associated to its being active. The notion of context is very general and subsumes all previous approaches. It is similar to the if/restriction mechanism, but allows scope information exposure. Context-aware aspects in Reflex can be implemented using either restrictions or activation conditions, offering different binding modes. Although by default thread global, if first-class activation is used as an implementation, thread locality can be obtained in context-aware aspects by reusing a thread-local semantics (as discussed previously).

Deploy block. CaesarJ supports a dynamic scoping mechanism that consists in restricting an aspect to the dynamic extent of a `deploy` block. For instance, `deploy(A){...}` implies that the aspect `A` is applied to all execution occurring within the dynamic extent of the block. In contrast with all the other scoping mechanisms we have seen so far, here the scope is *explicitly* embedded within the application code. Still, the definition has an *implicit* exit point (the end of the deploy block). Deploy blocks cannot be parameterized with values for access later down in

scoping mechanism	*action*	*definition*	*exp.*	*binding time/mode*	*thread local*
if/restriction	B	I	no	CT/S	M
receiver, target, etc.	B	I	yes	CT/S	M
1st-class activation	B	I	no	*depends*/D	M/A
cflow	B	I	yes	*depends*	A
tracematches	B	I	yes	CT/S	M/A
ctx-aware aspects	B	I	yes	CT/D	M
deploy block	B	EI	no	RT/S	A
on/off	B	EE	no	RT/D	M
classboxes	S	I	no	CT/S	A
mixin layers	S	EI	no	RT/D	A

B: behavioral – S: structural – I: implicit – EI: explicit start, implicit end – EE: explicit start and end – CT: compile-time – RT: run time – S: static – D: dynamic – M: manual – A: automatic

Fig. 3. Summary of the characterization of different dynamic scoping mechanisms.

the execution. The scope-aspect binding is done at runtime, but is indeed static: within the dynamic extent of the deploy block, the deployment cannot be undone and redone. Finally, the aspect is deployed locally to the thread executing the deploy block.

Explicit on/off. Some proposals support (de)activation of an aspect by explicit calls to `on/off`-like methods. In AspectS this can be done by executing `A install` or `A uninstall` at any time (dynamic binding at runtime). Less flexible versions of this feature can be implemented using first-class activation (as in Reflex) or even if/restriction mechanisms. In this approach, the dynamic scope to which an aspect is bound consists in the execution taking place in between a call to the `on` method and a call to the `off` method. Therefore both entry and exit points of the scope definition are explicitly defined. Like other low-level conditional mechanisms, thread locality has to be manually implemented.

Classboxes. Classboxes are a mechanism for dynamically-scoped structural *refinements*. Similar to open classes, refinements in classboxes make it possible to extend a class definition "from the outside" with new fields and methods, as well as extending methods with a mechanism similar to overriding in standard object-oriented programming. However, while changes made with open classes are globally visible, classboxes introduce a dynamic scoping mechanism based on import relations between modules: a refinement to a class is only visible for all execution that originates from a client of the module in which the refinement is defined. This property is checked at runtime: it can be extracted from the stack, and is therefore thread local. The definition of the scope of a refinement with classboxes is *implicitly* defined in the sense that the programmer does not explicitly says when the entry or exit points of a scope occur, but still requires *explicit* import declarations which are used by the runtime system to determine these points automatically. The binding is done at compile time, and is static. No context information can be exposed to the refinements.

Mixin layers. ContextL is a CLOS extension that introduces a mechanism of dynamic mixin layers, which are dynamically scoped. In ContextL, there is a notion of layers in which one can define structural refinements. These structural refinements can then be dynamically activated, by executing a set of expressions in the dynamic extent of a `with-active-layers` call. In other words, this mechanism can be seen as a combination of the deploy block mechanism of CaesarJ, but for structural actions like in classboxes. Therefore the definition of the scope is similar to the deploy block (explicit entry point, implicit exit point). However, as opposed to the deploy block a Caesar, a mixin layer can be temporarily deactivated, using a `with-inactive-layers`. Hence this mechanism supports dynamic binding at runtime. Thread locality is automatic, and, like the last three mechanisms presented, mixin layers cannot expose information to the refinements.

4. CONCLUSION AND DISCUSSION

We have described a frame of reference to compare different dynamically-scoped crosscutting mechanisms, including both structural and behavioral approaches. As a result of this study, differences between related mechanisms are clarified. For instance, the precise difference between an `if` pointcut and a first-class activation condition lies in the binding mode and possibly in the binding time (*e.g.* in the case of AspectS). Also, the contribution of context-aware aspects versus these conditional mechanisms comes to light: the possibility to expose information bound in the conditional to the action. Tracematching appears similar and therefore a deeper comparison of both mechanisms should be valuable. An interesting parallel can be made between the deploy block mechanism as in CaesarJ and the mixin layers of ContextL: apart from the fact that one is used for behavioral aspects and the other for structural changes, they actually differ with respect to the binding mode (the binding time is the same). More comparative results appear directly from looking at Fig. 3.

Among the suggestions for future development in the area, it seems noteworthy that all mechanisms that are explicitly defined (deploy block, on/off, classboxes, mixin layers) actually lack the context information exposure feature. As a matter of fact, such a feature does make sense for these mechanisms. This remark also holds for first-class activation.

Finally, it seems that a discussion about the different definition mechanisms is valuable. Most mechanisms embed the scope definition outside of the base code (implicit). The explicit ones require intrusive specification (*i.e.* base code embeds the scope boundaries definition) but prove useful in some cases, thanks to the dynamicity of the binding that is thereby obtained. Of course, it is usually feasible to turn an explicit definition into an implicit one (by using an aspect!).

With respect to tractability, it is clear that dynamic scoping requires more effort from the programmer. In the case of explicit scope definitions, both deploy blocks and mixin layers as in ContextL have a clear scope, in the sense that the end of the scope is implicitly associated to the end of the lexical structure that specifies the begin of the scope (*i.e.* the closing } of a deploy block). Conversely, the on/off mechanism is much less "structured", in the sense of structured programming vs. GOTO-like programming [Dijkstra 1968]. If many explicit on/off statements are used in various places of an application, it can easily turn out to be impossible for a programmer to precisely foresee when an aspect will actually apply. On the other hand, it is true that in some cases, such an explicit mechanism turns out to be useful. At least for two reasons: first of all, because it is the lowest-level mechanism, it can be used to experiment with higher-level abstractions like the ones we discussed in this paper; second, because in some cases, like context-oriented programming [Costanza and Hirschfeld 2005], it seems necessary to be able to align aspect application to events that can happen at any time, like a user logging out of a system. If we make the analogy with the history of GOTO-like programming, then we may foresee that these kinds of (too?) flexible schemes may be bound to disappear in the light of a sound compromise. But this sound compromise is yet to be found.

In any case, dynamically-scoped aspects definitely challenge traditional visions of program understanding, much more than AOP in its implicit and lexically-scoped flavor –which is already a challenge compared to more traditional paradigms–.

Acknowledgments. The author thanks the following colleagues for the interesting discussions about one or more of the presented mechanisms: Alexandre Bergel, Eric Bodden, Pascal Costanza, Marcus Denker, Brecht Desmet, Kris Gybels, Robert Hirschfeld, and Theo D'Hondt.
É. Tanter is partially financed by the Milenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

REFERENCES

Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. 2005. Adding trace matching with free variables to AspectJ. See OOPSLA 2005 [2005], 345–364. ACM SIGPLAN Notices, 40(11).

Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K. 2006. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*. Lecture Notes in Computer Science, vol. 3880. Springer-Verlag, 135–173.

Bergel, A., Ducasse, S., and Nierstrasz, O. 2005. Classbox/J: Controlling the scope of change in Java. See OOPSLA 2005 [2005], 177–189. ACM SIGPLAN Notices, 40(11).

Bergel, A., Ducasse, S., and Wuyts, R. 2003. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of the Joint Modular Languages Conference (JMLC'03)*. Lecture Notes in Computer Science, vol. 2789. Springer-Verlag, 122–131.

Costanza, P. and Hirschfeld, R. 2005. Language constructs for context-oriented programming – an overview of ContextL. In *ACM Dynamic Language Symposium (DLS 2005)*. San Diego, CA, USA.

Dijkstra, E. W. 1968. Go To statement considered harmful. *Communications of the ACM 11,* 3 (Mar.), 147–148.

Douence, R., Fradet, P., and Südholt, M. 2004. Composition, reuse and interaction analysis of stateful aspects. See Lieberherr [2004], 141–150.

DOUENCE, R., FRADET, P., AND SÜDHOLT, M. 2005. Trace-based aspects. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, Eds. Addison-Wesley, Boston, 201–217.

ELRAD, T., FILMAN, R. E., AND BADER, A. 2001. Aspect-oriented programming. *Communications of the ACM 44,* 10 (Oct.).

HILSDALE, E. AND HUGUNIN, J. 2004. Advice weaving in AspectJ. See Lieberherr [2004], 26–35.

HIRSCHFELD, R. 2002. AspectS – aspect-oriented programming with Squeak. In *International Conference NetObjectDays on Components, Architectures, Services, and Applications for a Networked World*, M. Akşit, M. Mezini, and R. Unland, Eds. Lecture Notes in Computer Science, vol. 2591. Springer-Verlag, 216–232.

HIRSCHFELD, R. AND COSTANZA, P. 2005. Extending advice activation in AspectS. In *2nd European Interactive Workshop on Aspects in Software (EIWAS 2005)*. Brussels, Belgium.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. 2001. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, J. L. Knudsen, Ed. Number 2072 in Lecture Notes in Computer Science. Springer-Verlag, Budapest, Hungary, 327–353.

LADDAD, R. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Press.

LIEBERHERR, K., Ed. 2004. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004).* ACM Press, Lancaster, UK.

MASUHARA, H. AND KAWAUCHI, K. 2003. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03).* Lecture Notes in Computer Science, vol. 2895. 105–121.

MASUHARA, H., KICZALES, G., AND DUTCHYN, C. 2003. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003)*, G. Hedin, Ed. Lecture Notes in Computer Science, vol. 2622. Springer-Verlag, 46–60.

OOPSLA 2005 2005. *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005).* ACM Press, San Diego, California, USA. ACM SIGPLAN Notices, 40(11).

REDMOND, B. AND CAHILL, V. 2002. Supporting unanticipated dynamic adaptation of application behavior. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, B. Magnusson, Ed. Number 2374 in Lecture Notes in Computer Science. Springer-Verlag, Málaga, Spain, 205–230.

TANTER, É., GYBELS, K., DENKER, M., AND BERGEL, A. 2006. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, W. Löwe and M. Südholt, Eds. Lecture Notes in Computer Science, vol. 4089. Springer-Verlag, Vienna, Austria, 227–242.

TANTER, É. AND NOYÉ, J. 2005. A versatile kernel for multi-language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, R. Glück and M. Lowry, Eds. Lecture Notes in Computer Science, vol. 3676. Springer-Verlag, Tallinn, Estonia, 173–188.

TANTER, É., NOYÉ, J., CAROMEL, D., AND COINTE, P. 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 2003)*, R. Crocker and G. L. Steele, Jr., Eds. ACM Press, Anaheim, CA, USA, 27–46. ACM SIGPLAN Notices, 38(11).

VANDERPERREN, W., SUVEE, D., CÍBRAN, M. A., AND DE FRAINE, B. 2005. Stateful aspects in JAsCo. In *Proceedings of Software Composition (SC 2005).* Lecture Notes in Computer Science, vol. 3628. Springer-Verlag, 167–181.