Dynamic Virtual Join Point Dispatch

Christoph Bockisch   Michael Haupt   Mira Mezini
Software Technology Group
Darmstadt University of Technology
Darmstadt, Germany

{bockisch,haupt,mezini}@informatik.tu-darmstadt.de

ABSTRACT

Conceptually, join points are points in the execution of a program and advice is late-bound to them. We propose the notion of *virtual join points* that makes this concept explicit not only at a conceptual, but also at implementation level. In current implementations of aspect-oriented languages, binding is performed early, at deploy-time, and only a limited residual dispatch is executed. Current implementations fall in the categories of modifying the application code, modifying the meta-level of an application, or interacting with the application by means of events—the latter two already realizing virtual join points to some degree. We provide an implementation of an aspect-oriented execution environment that supports truly virtual join points and discuss how this approach also favors optimizations in the execution environment.

1. INTRODUCTION

Procedure calls and declarations are well-known mechanisms for binding meaning to points in the execution of a program. In [28], pointcuts are presented as the natural successors of these mechanisms. In fact, they bear a conceptual resemblance to virtual methods in object-oriented (OO) languages in the sense that both are late-bound to meaning. Whenever a virtual method is called, some predicate is evaluated. The expressiveness ranges from a predicate on the dynamic type of the receiver in "traditional" method dispatch [18] to a predicate on the types of receiver and arguments in multi-dispatch [15], or, more radically, to an arbitrary predicate on dynamic values in predicate dispatch [30]. Similarly, when a join point occurs, pointcuts are, at least conceptually, evaluated the to determine the meaning to bind to the join point: the advice of those pointcuts that match are executed; if no pointcut maches, the join point is executed unadvised.

This conceptual view is reflected in formalisms for OO and aspect-oriented (AO) languages [14, 27, 29] and often called the "direct semantics" of AOP in informal discussions.

However, it is not reflected in most current AO language implementations. The way in which such implementations perform deployment is normally invasive with regard to application code: application bytecodes are manipulated and advice dispatch logic is woven in at so-called *join point shadows* [29].

Hence, there is a discrepancy between the implementation of AO programming languages and modern OO programming languages, such as Java [20], Smalltalk [38], and Self [40]. In these languages, the compiler does not statically undertake any dispatching steps for method calls, it merely determines the dispatch strategy. In Java, e.g., this may be dispatch using a virtual method dispatch table [18] or, for interface methods, searching the method in the complete super type hierarchy [2]. However, all method calls remain virtual after compilation, i.e., their binding is per default determined by evaluating run-time predicate functions. It is the virtual machine that dynamically applies optimizations and decides to treat some of the method calls as non-virtual ones. In doing so, it also takes care of applying the right guards responsible for "watching" dynamic class loading which potentially invalidates previously taken decisions about the virtuality of method calls [5].

This is different with traditional aspect language processors. So far, only program elements that are identified as join point shadows by the weaver through some sort of static analysis of the code "get the chance" to ever become join points at run-time. There are several problems with this approach.

On the one hand, dynamic aspect deployment becomes expensive since the manipulation of application bytecodes and their subsequent reinstallation into the execution environment are time-consuming. Approaches that exhibit very fast dynamic deployment behaviour by preparing join point shadows with hooks and wrappers usually suffer from the performance overhead of this footprint [21].

On the other hand, dynamic class loading is poorly supported. It is, for example, possible to statically optimize a `cflow` pointcut to the best possible degree using sophisticated analyses [9]. However, when classes are dynamically loaded which have not been available during static analysis the optimizations fail and the aspect may not be executed correctly anymore.

Many existing dynamic weaving approaches do not provide satisfying solutions to these problems. Most of them basically employ the same approach as static weavers: join point shadow retrieval and advice weaving remain the same, only deployment is delayed. Dispatch is still actually applied

at deployment time by means of inserting dispatching logic into application code. In the more optimized approaches, this is frequently performed by dynamic code modification, e. g., in AspectWerkz [8], which may, as in Steamloom [22], even be supported by the virtual machine itself. Such approaches frequently exhibit long weaving times [12, 21] and, hence, slow deployment.

The question to answer is whether there is a counterpart, applicable to AO language implementations, to the techniques used by other language implementations to optimize away dispatch predicates. In our recent work [10, 11] (cf. Sec. 3) we map virtual join points to virtual methods. Preliminary results show that OO optimization techniques are beneficial in providing truly virtual join points and can be reused to a considerable degree. We analyzed which kinds of late binding are possible or will be possible in the future. In the related work, we discuss current approaches which already support virtual join points, but which do not facilitate optimizing the dispatch.

This paper is structured as follows. The idea of virtual join points is presented in the next section, along with considerations as to which parts of execution environments need to be taken into account when devising an optimized implementation of the idea. We present a first prototype implementation of virtual join points in Sec. 3. In Sec. 4, reflective and event-based systems are discussed with regard to how they facilitate dispatch at join points. AOP implementations that exploit such mechanisms are also briefly presented. The benefits of virtual join points in general, and of the presented implementation in particular are summarized in Sec. 5, along with a brief discussion of future work.

2. VIRTUAL JOIN POINTS

In an ideal aspect-oriented programming language, advice should be late-bound to join points at run-time. In the following, we will use the term *join point* to denote not only points in execution-time but also the according implementation, i.e., the action carried out at this point. When an advice is bound to a join point, its implementation consists of the advice execution as well as the original action.

To facilitate late binding of join points, some kind of dispatch-mechanism is required; this is similar to virtual methods in object-oriented programming languages. To motivate this claim and further explain it, we shortly describe virtual method dispatch.
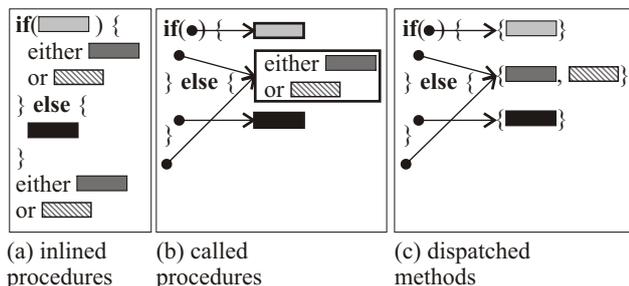

(a) inlined procedures

(b) called procedures

(c) dispatched methods

Figure 1: Sketch of code that uses (a) no procedures at all, (b) procedures that are early-bound, (c) virtual methods that are late-bound.

Part (a) in Figure 1 shows how a program looks like that uses no procedures at all: the code of different concerns ap-

pears sequentially, possibly several times, in the program. When we introduce procedures into the program, each concern is refactored into one procedure and the original code is replaced by a call to the procedure, as seen in part (b). However, the procedure is still statically bound to the call site and there is no variability of which procedure is called at run-time. Finally, in part (c), we show what the program looks like when virtual methods are used instead of procedures. At each call site, there is a set of potential target methods—which one is executed at run-time is only decided just before the method is called. The process of making this decision is called dispatching.

In the first programming style, certainly, every program can be written. But the style has the disadvantage that code is duplicated and consequently not well-modularized. Further, to allow flexibility of which code is executed at run-time, dispatching must also be coded by the programmer.

The second programming style improves modularity by refactoring duplicated code into single procedures, while dispatching is still coded in the application. Finally, with virtual methods, the flexibility of late binding the concrete method to be executed is provided implicitly by the execution environment.

We see a tight resemblance between the concepts of procedures and virtual methods, and the concept of join points. In fact, we claim we can seamlessly replace *procedure* and *method* in Figure 1 with *join point*, in the sense of "action" to be executed. If a program is not written in an aspect-oriented programming language, it looks like (a)—crosscutting concerns are tangled with the application and scattered over it. Aspect-oriented programming languages allow to localize these concerns, but current implementations of these languages for the most part bind advice to join points *early*. The bytecode to which these implementations compile aspect-oriented programs resembles part (b) from the figure. Although the conditional logic is generated by the AO language implementation, it is part of the application code. Having a dispatch for join points also in the compiled code as in part (c) should be the goal of AO language implementations.

A logical consequence is to realize join points as *virtual* join points. The main difference to methods is that application programmers do not have to explicitly insert a call to the join point, as they have to for virtual methods. Instead, the *execution environment* implicitly turns join points into virtual join point "calls" and inserts the original implementation into the set of potential targets. If no advice applies to the join point, there is only one potential target. This is comparable to a virtual method that is not overwritten.

The virtual join point "call" is dispatched at run-time and one target is selected according to the current run-time state of the application. The run-time state, e. g., depends on which aspects are deployed in which contexts (e. g., an aspect may be deployed only for one thread). Consequently, the concrete implementation of a join point, advised or not, is late-bound.

Languages using static weaving, like AspectJ, already provide means to state run-time conditions that must be satisfied in order for an advice to be executed. One example is the `cflow` pointcut designator. This is realized by inserting what is called *residual dispatch* [27] before the advice. Residual dispatch only executes that part of the dispatch which is not resolved statically. Also, it is only inserted at code loca-

tions that have been determined by a weave-time analysis. This kind of realizing dynamic pointcuts only works if all assumptions made at weave-time are still true at run-time. If an assumption is broken, e. g., by unforeseen dynamic class loading, the aspect is not executed correctly.

Now that we have motivated the need of dynamically dispatching join points, we still have to describe how powerful dispatch has to be. In object-oriented programming languages like Java or Smalltalk, a method call is dispatched only after the receiver object's type [20, 38]. This can be be realized by using a dispatch table.

Another kind of dispatch is multi-dispatch [15] where the receiver's type and all the arguments' types are taken into account. To realize this, an extended table is needed which has not only a mapping from one type to a method but from multiple types. Finally, there is the most general notion of predicate dispatch [30]. In this dispatching scheme, an arbitrary predicate is attached to a method: if it evaluates to true, the method is executed.

Predicate dispatch surely is the most powerful variant, but evaluating arbitrary predicates is costly and hard to optimize. Dispatching after one run-time object is very efficient (because virtual method tables can be used) but does not offer sufficient power. To prove this and to show how powerful the dispatch function must be, we give some examples from existing programming languages where advice execution depends on run-time conditions.

AspectJ [6] provides the dynamic pointcut designators `cflow`, `target`, `this` and `args`, which specify the current control flow, respectively the dynamic type of the receiver, active or argument objects. Consequently, the dispatch has to take these into account. Other than the receiver, active and argument objects, the control flow is no first-class object in Java. AspectJ's implementation for the `cflow` pointcut designator is to trace the entering in / exiting from the control flow and store the result in a global variable that is accessed in the residual dispatch.

Other AOP languages like CaesarJ [3], Steamloom [22], JAsCo [42, 41], Association Aspects [37], PROSE [35] or EOS [36] also allow for deploying an aspect, e. g., only in certain threads or for certain objects. As a result, the current thread can be a dimension of dispatch, as well as the active or receiver objects themselves—not only their types.

We also see the state of which aspects are currently deployed as a dimension of dispatch. Many current AOP languages allow to deploy and undeploy aspects at run-time. Consequently, if an aspect declares advice for a join point, the join point must be dispatched differently, whether the aspect is deployed or not.

The presented dispatch dimensions are sufficient to realize current AOP languages. Other dimensions are perceivable, though, that hint at the capabilities of upcoming and future AOP languages. If, for example, a pointcut language takes into account the history of execution [41, 1] or the interconnections of objects on the heap [32], dispatch dimensions come into scope that are laborious to implement with currently existing approaches. The generalized concept of virtual join point dispatch delivers a more powerful basis on which such languages can be implemented.

3. ENVELOPE-BASED WEAVING

We have developed a new technique for dynamically deploying aspects, called *envelope-based weaving*, by adapting

techniques used for virtual methods. In this section, we discuss this technique as a first step towards truly virtual join points as described in Sec. 2.

Envelope-based weaving [10, 11, 12] was realized for the Java language with the additional support for dynamically deploying AspectJ-like pointcut-and-advice. The implementation is based on IBM's Jikes Research Virtual Machine (Jikes RVM) [26]. In this section, we will first present some standard techniques of OO virtual machines which we extended to support envelope-based weaving. Afterwards, we outline our implementation of envelope-based weaving as an optimizing support for special cases of virtual join points. We conclude this section by discussing the analogy between envelopes and virtual join points. Sec. 5 contains an evaluation of this approach.

Jikes is completely based on just-in-time compilation. That is, when a method is to be executed, Jikes first generates machine code for it and the method is natively executed. Jikes may decide to apply optimizations when compiling a method, the most important of which is inlining. Inlining can only be applied when the JIT compiler exactly knows which implementation of a method will allways be executed at a call site.

It is also possible to inline a method, even if at JIT compile-time it is only assumed that there will be only one possible target method for the rest of the execution, and this assumption may turn wrong. In Java, this happens when a new class is loaded overwriting the previously assumed unique target method. When a method is inlined based on an assumption that may get invalidated, the inline location is *guarded* [17], which facilitates undoing the inlining optimization.

A Java program that is executed by our modified JVM is transformed into *envelope style*, which means that each join point (i. e., method call and field read or write in our approach) is replaced by a call to an envelope—an envelope is realized as a standard method in our current implementation. One envelope is generated for each defined method and two for each defined field: one for read and one for write access. The envelope consists of the action that the envelope call replaces, i. e., a call to the enveloped method or reading/writing the enveloped field.

When an aspect is deployed, we can determine which envelopes are affected and weave the aspect's advice into these envelopes. Conceptually, we create new envelopes and make sure that the dispatch selects the new envelopes as long as the aspect is deployed. In the implementation this is realized by replacing the affected envelopes' bytecodes with the advised ones and storing the unadvised bytecodes as a backup.

For an advised version of an envelope to become effective, we re-JIT-compile the envelope and install the new machine code in the dispatch table. If the old version of the envlope is inlined somewhere, we can determine the affected inline locations and undo the inlining, so that the advised envelope will be subsequently executed. To facilitate this, envelopes are always guarded when they are inlined. We undo inlining in a way that allows the virtual machine to re-apply the inlining optimization by inlining the new version of the envelope. This way, deploying an aspect does not degrade the optimization state of the application's machine code.

Currently, envelopes are not real virtual join points yet, but they provide a first step in this direction. A subset of the described dimensions of dispatch is realized in an opti-

mized way. Supported dimensions are the receiver type and aspect-deployment state. The receiver type dimension is supported by the way we generate envelopes: they basically replace the original methods. Consequently, the envelope of a polymorphic method is also polymorphic. The dimension of aspect-deployment state is realized by modifying the dispatch of envelopes when an aspect is deployed as discussed above.

4. RELATED WORK

In this section, we briefly present and discuss AOP implementations that have at least limited support for virtual join points. Approaches that exhibit such support are met in two families of AOP implementations, namely systems based on reflection and meta-model manipulation, and event-based systems.

4.1 Reflective Approaches

A reflective system has the ability to represent application structures as first-class entities and make them available for manipulation. Depending on the possible degree of reflective access to the meta-level in a particular system, extended dispatching behaviour can be achieved by selectively updating data structures that serve as input for dispatch mechanisms, or the dispatching mechanisms themselves.

Hence, reflective systems allow for virtual join point dispatch, since virtually all kinds of information can be taken into account for dispatching. The ability of a reflective system to reify entities facilitates powerful dispatch mechanism implementations. It depends on the degree of reflective access what granularity of a join point model an AOP implementation based on reflection can achieve.

AspectS. AspectS [23, 7] is an AOP language implemented in the Squeak Smalltalk environment [25, 39]. AspectS is in fact a framework implemented on top of Smalltalk reflection to support AOP. It does not define any new language constructs. AOP support is implemented using only Smalltalk's reflective capabilities.

AspectS' join point model is very simple: only message receptions are supported as join points. This does not restrict expressiveness, since message passing is *the* core mechanism in Smalltalk anyway: method invocations as well as member variable accesses are implemented using messages.

All weaving in AspectS takes place at the meta-level. When a message implementation is decorated with advice functionality, its entry in the corresponding class' method dictionary is modified to reference a *method wrapper*. The wrapper invokes advice functionality and yields control to the original implementation, or to subsequent wrappers, if multiple advice apply.

The application of dispatch modifications to meta-level structures as met in AspectS augments the dispatching logic in the form of Smalltalk's original lookup mechanism for late-bound methods. The lookup mechanism itself is altered, along with the data that it operates on. When several wrappers are attached to a join point, the so-called *wrapper chain* must be iterated over, checking for each particular wrapper's applicability using conditional logic contained in the wrappers. Aggressive optimizations as usually found in sophisticated OO language implementations are not applicable due to the implementation of the iteration over the wrapper chain as a method with full computational power.

4.2 Event-Based Approaches

In an event-based system, the environment provides means to signal the occurrence of certain situations as events to which a running application may react. Like with reflective systems, this takes place without the need for modifying the application itself. In this case, the application's metamodel is also left untouched: all augmentation of behaviour is solely achieved through the registration of events and the definition of reactions thereto.

Depending on the flexibility of the event model, various degrees of granularity can be achieved in signalling situations. Event-based systems reach far beyond the point of providing means to register actions to take place when certain instructions are executed. Event-condition-action (ECA) rules [16] as met, e.g., in active database systems are able to react to complex situations depending on various sources of events.

It has been observed that ECA rules and aspect-oriented programming bear some resemblance [19, 13]: a join point is signalled by an event, to which a condition is applied as a dispatch predicate. If the predicate matches, an action is executed as an advice. An implementation of an event-based system can thus serve as the basis for an AOP language implementation.

PROSE. The architectural characteristics of PROSE [31, 33, 34, 35] exhibit a strong resemblance to event-based systems. PROSE has a two-layered architecture. The *AOP engine layer* takes care of aspect management, advice execution and other high-level tasks that are independent of a specific implementation technique of low-level mechanisms. The latter are provided by the *execution monitor layer*, which is responsible for join point signalling. It notifies the AOP engine of join point occurrences, to which the AOP engine then can react by, e.g., executing advice functionality. The AOP layer also explicitly asks the execution monitor layer to (un)register specific join points as needed by the aspects managed by it.

While the AOP layer is quasi-standardized, there exist, in the present version 1.3.0 of PROSE, several different implementations of the execution monitor that directly reflect on the basic mechanisms used [31, 21]. Of these implementations, the one called *debugger-based weaving*, is most interesting with regard to the discussion of PROSE as an eventbased system. In this implementation, the JVM's standard debugger API can be employed to reify join points as debugger breakpoints. Hence, a core service of the virtual machine is used to signal join points.

The debugger-based weaving approach comes close to an "ideal" solution for AOP. Exploiting the virtual machine's debugger infrastructure is certainly not satisfactory, but this variant of the PROSE execution monitor shows that a noninvasive event-based system can operate as the basis of an AOP run-time environment.

While the execution monitor takes over the responsibility of signalling events, the AOP engine layer in PROSE implements the actual dispatching logic applied at join points. Since the AOP engine is implemented in Java, it has full computational power: arbitrary information and mechanisms can be exploited to dispatch advice, which is a strong advantage. The downside is that it hinders optimization because the dispatch mechanisms are implemented with full computational power and must be executed in the given way for

all join points, which leads to poor performance [21].

5. CONCLUSION AND FUTURE WORK

We claim that join points should be considered virtual not only because dispatching them is more implicit and natural; it is also for reasons of performance. Realizing dynamic dispatch is much more efficient than re-weaving the whole program code of an application. We believe that this is important in order to facilitate the regular use of dynamic aspect deployment.

In our implementation of the envelope-aware Jikes RVM we exploited and adapted the optimizations of virtual methods to provide a first implementation of virtual join points. We could also prove that supporting this concept with VM-level optimizations speeds up deployment and actually lifts it to a level where regular use is feasible. We measured how long dynamic deployment takes, using the SPECjvm98 benchmark suite. In each benchmark, we dynamically deployed an aspect which advises calls to all public methods, i. e., the envelopes of all public methods are affected. With envelope-based weaving, this takes at most one millisecond. We measured the same for other AO execution environments that support dynamic deployment. The same deployment took between 1,442 and 9,864 milliseconds for AspectWerkz, between 143 and 1,344 milliseconds for PROSE and between 17 and 4,363 milliseconds for Steamloom, depending on the executed benchmark. From these results it is visible that the dispatch approach, which is partly also followed by PROSE, is promising.

Previous measurements [12] have shown that the event-based approach as followed by PROSE does not offer the execution environment the possibility to optimize the execution of aspect-oriented programs. In PROSE, the dispatch is realized as Java methods which is on the one hand powerful, but hard to optimize on the other hand. As seen in [12], PROSE is slowed down by several orders of magnitude when dispatch must be executed. There is nearly no slowdown for systems like AspectJ, AspectWerkz or Steamloom, so hard-wiring dispatch seems to offer the execution environment the opportunity for optimiziations, however, at the cost of reduced flexibility.

With envelope-based weaving, we have shown that it is possible to combine the best of both worlds. We realize dispatch at VM level which allows the VM to optimize it. Virtual join points supported at VM level provides a great flexibility as well as performance.

We will continue with the implementation of envelope-based weaving and follow our concept of virtual join points more consequently in the future. We are also working on supporting more dimensions of dispatch. Representing the run-time conditions based on which dispatch is performed as first-class objects is beneficial for an efficient implementation, as dispatch tables can be used in this case. This is why one of our areas of research is to find or provide first-class objects for this purpose.

For control flows, e. g., a first-class representation of method activations, i. e., call stack frames, can be used. Such an activation can contain a dispatch table which is used to look up the appropriate implementation of envelopes.

Another area of research is providing optimizations for more powerful dispatch mechanisms. There are other kinds of guards [4] than those we are already making use of in the implementation of envelope-based weaving. Most no-

tably, we will investigate how *thin guards* [5] can be used to optimize virtual join point dispatch. We will also investigate which new kinds of guards can be realized and how the technique of polymorphic inlining [24] can be exploited.

6. REFERENCES

[1] C. Allan et al. Adding Trace Matching with Free Variables to AspectJ. In *Proc. OOPSLA 2005*, pages 345–364. ACM Press, 2005.

[2] Bowen Alpern, Anthony Cocchi, Stephen J. Fink, David Grove, and Derek Lieber. Efficient implementation of java interfaces: Invokeinterface considered harmless. In *Conference on Object-Oriented*, pages 108–124, 2001.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of caesarj. *Transactions on AOSD I (to appear)*, LNCS 3880, 2006.

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proc. IEEE*, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation.

[5] M. Arnold and B. G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *Proc. ECOOP 2002*, volume 2374 of *LNCS*, June 2002.

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

[7] AspectS Home Page. `http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/`.

[8] AspectWerkz Home Page. `http://aspectwerkz.codehaus.org/`.

[9] P. Avgustinov et al. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–128. ACM Press, 2005.

[10] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting vm techniques for seamless aspect support. submitted for review to the conference onf Virtual Execution Environments '06.

[11] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Envelope-aware Virtual Execution Environment (EVE). http://www.st.informatik.tu-darmstadt.de/EVE.

[12] C. Bockisch and M. Haupt and M. Mezini and R. Mitschke. Envelope-based Weaving for Faster Aspect Compilers. In *Proc. NetObjectDays 2005*. GI, 2005.

[13] M. Cilia, M. Haupt, M. Mezini, and A. P. Buchmann. The Convergence of AOP and Active Databases: Towards Reactive Middleware. In F. Pfenning and Y. Smaragdakis, editors, *Proc. GPCE 2003*, volume 2830 of *LNCS*, pages 169–188. Springer, 2003.

[14] C. Clifton, G. Leavens, and M. Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages, 2003.

[15] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

[16] U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, volume 334 of *Lecture Notes In Computer Science*, pages 129–143. Springer, 1988.

[17] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *Proc. ECOOP 1999*. Springer, 1999.

[18] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[19] R. E. Filman and K. Havelund. Source-Code Instrumentation and Quantification of Events. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Workshop (at AOSD 2002)*, pages 45–49, 2002.

[20] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1979.

[21] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Darmstadt University of Technology, Computer Science Department, 2006. to appear.

[22] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In *Proc. VEE 2005*. ACM Press, June 2005.

[23] R. Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*, pages 216–232. Springer, 2003.

[24] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches, 1991.

[25] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *Proc. OOPSLA 1997*, pages 318–326. ACM Press, 1997.

[26] The Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[28] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In A. Black, editor, *ECOOP 2005 - Object Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings*, volume 3586 of *LNCS*. Springer, 2005.

[29] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In G. Hedin, editor, *Proc. CC 2003*, volume 2622 of *LNCS*, pages 46–60. Springer, 2003.

[30] T. Millstein. Practical predicate dispatch. In *2004 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*. ACM Press, 2004.

[31] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. `http://www.iks.inf.ethz.ch/publications/publications/files/PROSE-ASMEA05.pdf`.

[32] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In A. Black, editor, *ECOOP 2005 - Object Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings*, volume 3586 of *LNCS*. Springer, 2005.

[33] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In G. Kiczales, editor, *Proc. AOSD 2002*. ACM Press, 2002.

[34] A. Popovici, T. Gross, and G. Alonso. Just-in-Time Aspects. In *Proc. AOSD 2003*. ACM Press, 2003.

[35] PROSE Home Page. `http://prose.ethz.ch`.

[36] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[37] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD*, pages 16–25, 2004.

[38] Smalltalk. http://www.smalltalk.org.

[39] Squeak Home Page. `http://www.squeak.org/`.

[40] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[41] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. `http://ssel.vub.ac.be/jasco/media/sc2005.pdf`.

[42] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in jasco. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM Press.