A Machine Model for Aspect-Oriented Programming

Michael Haupt[1] and Hans Schippers[2⋆]

[1] Software Architecture Group
Hasso Plattner Institute for Software Systems Engineering
Potsdam, Germany
[2] Formal Techniques in Software Engineering
University of Antwerp, Belgium
michael.haupt@hpi.uni-potsdam.de, hans.schippers@ua.ac.be

Abstract. Aspect-oriented programming languages usually are extensions of object-oriented ones, and their compilation target is usually the (virtual) machine model of the language they extend. While that model elegantly supports core object-oriented language mechanisms such as virtual method dispatch, it provides no direct support for core aspect-oriented language mechanisms such as advice application. Hence, current implementations of aspect-oriented languages bring about insufficient and inelegant solutions. This paper introduces a lightweight, object-based machine model for aspect-oriented languages based on object-oriented ones. It is centered around delegation and relies on a very dynamic notion of join points as loci of late-bound dispatch of functionality. The model is shown to naturally support an important number of aspect-oriented language mechanisms. Additionally, a formal semantics is presented as an extension to the object-based δ calculus.

1 Introduction

The progress of the aspect-oriented programming (AOP) paradigm [43, 27] has spawned a wide variety of AOP languages and corresponding implementations [13]. Such languages are usually formulated as extensions of object-oriented "base" programming languages; and they are usually implemented by expressing AOP core mechanisms (such as advice application at join points) [22] in terms of the base language mechanisms.

For example, AspectJ [42, 4] is an extension of Java [31, 47]. AspectJ compilers generate Java bytecodes. The same holds for other Java-based AOP languages and systems [3, 67, 52].

In other words, the machine models targeted by compilers for object-oriented and aspect-oriented programs are the same. AOP languages' core mechanisms are *transformed* into a representation using only object-oriented mechanisms, because those are the only ones that the target machine understands. Consequently,

⋆ Research Assistant of the Research Foundation, Flanders (FWO)

representations of aspect-oriented core mechanisms tend to be "verbose" in their object-oriented executable representation, as workarounds have to be found for mechanisms that cannot be directly expressed by the target machine.

For instance, regard the application of a before advice at a method execution join point. In AspectJ, it is transformed into two method calls that are inserted at the beginning of each affected method: one call to retrieve an appropriate instance of the aspect, and one to invoke the advice. The latter is implemented as a method in a class representing the aspect [13, 35].

The transformation of aspect-oriented code to fit an object-oriented target machine model introduces a semantic gap between the language's expressions and their realisation. It is especially apparent when regarding the target representation of *join points*. Join points are well-defined points in the execution graph of a running application [42, 43, 27]: points at which functionality defined in aspects is made effective. Transformation of aspect-oriented code to an object-oriented target machine model usually represents them in the form of *join point shadows* [35]: *locations in application code* where join points potentially occur at run-time.

Most AOP language implementations follow an approach centered around this notion, i. e., they regard applications during weaving solely in terms of their static representation in *code*. This contradicts the accepted view on join points as being inherently *dynamic*. In essence, conceptual and technical views on join points and the realisation of attaching advice functionality to them are unnaturally different: dynamic properties are ultimately expressed using static means, such as code locations.

The aforementioned semantic gap has been observed earlier [9] and led to the development of dedicated virtual-machine level support for AOP in the form of the Steamloom VM [9, 34, 33]. While Steamloom set out to bridge the gap, it has achieved less. On the one hand, several techniques dedicated to offer explicit support for core AOP mechanisms have been devised [33, 10, 7]. On the other hand, Steamloom operates at bytecode level, still expressing AOP mechanisms targeting an object-oriented machine model. Recent advances in virtual machine-level weaving support [7] still follow this direction.

To effectively bridge the gap, it is required to devise an *aspect-oriented machine model* that can directly be targeted by AOP language compilers. This paper's contribution is a first version of such a model.

As the foundation for the model, we propose the notion of *virtual join points*[1]. The notion of a join point as a point in the execution flow of a program suggests to regard it as a *locus of late binding*. This view has been mentioned several times [42, 49, 14] but, to the best of our knowledge, not been consequently adopted in implementations so far.

At every join point—seen as a locus of late binding of functionality or value to messages—, dispatch takes place, even though it leads, in most cases, to the execution of the join point's "original" functionality. Dispatch is oriented along multiple dimensions, i. e., relies on one or more different properties from the

[1] Some of the core ideas have been formulated in a workshop paper [8].

program state at the time a particular join point is reached. One such dimension is equivalent to virtual method dispatch, where the dynamic type of the object receiving a message send determines the operation to be executed.

In AOP, dispatch dimensions are manifold, and numerous dynamic properties come into question, e. g., the current control flow in case of `cflow`, the current thread, sending/receiving instance, or others. Of course, static properties, such as the message sent in the case of `call` or `execution` join points are also viable candidates.

Viewing join points as loci of late binding yields a consistent point of view, enabling a fresh view on the execution of aspect-oriented programs, and on the implementation of execution environments for aspect-oriented programming languages. If a running application is regarded as a sequence of join points [42], adopting the aforementioned notion suggests to also regard it as a series of late-binding events, of virtual functionality dispatch. In the following, we will elaborate on AOP implementations and how they adhere to the new view on join points mentioned above.

Based on the notion of virtual join points, we propose a machine model for AOP languages called *delegation-based AOP* that faithfully obeys the view on join points as loci of late binding. It is formulated as an extension of a prototype-based object model and uses delegation to achieve late binding.

It is important to note that the proposed model is indeed the core of a *machine model* for AOP; it is *not* a programming language. The model can be thought of as the internal representation of "AOP assembler" in a (virtual) machine with dedicated direct support for AOP mechanisms.

The structure of this paper is as follows. In the next section, we introduce the concept of virtual join points in detail. After that, in Sec. 3, we present the execution model of delegation-based AOP in a purely prototype-based setting, as well as a description of how the model can be extended to support, at the language-implementation level, AOP in class-based languages. An operational semantics is presented as an extension to the δ calculus [2] in Sec. 4. Sec. 5 discusses related work. Finally, Sec. 6 summarises the paper and outlines future work.

2 Virtual Join Points

A join point is an inherently dynamic element of a running application, and it is a locus of late binding. To facilitate late binding at join points, a 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.

Fig. 1(a) shows a program using no procedural abstraction at all: the code of different concerns appears sequentially, possibly several times, in the program. A choice between two concerns—depicted by the "either/or" alternative—is, in such an approach, usually implemented using an `if` statement.

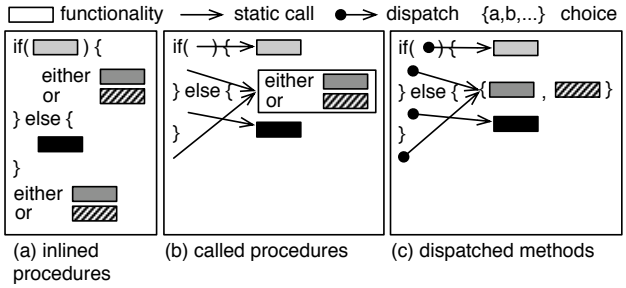

Fig. 1. Sketch of code that uses (a) no procedures, (b) procedures that are early-bound, (c) virtual methods that are late-bound.

When procedures are introduced 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. The concern choice is, in the procedure, also still explicitly represented.

Finally, in part (c), we show the program's shape 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 explicit implementation of the either/or choice has vanished and is replaced by an implicit process called *dispatching*.

The first programming style's disadvantage is that code is replicated and consequently not well modularised. The second style improves modularity by refactoring replicated code into procedures, while dispatching is still coded in the application. Finally, with virtual methods, the flexibility of late-binding is provided implicitly by the execution environment.

We see a close resemblance between the concepts of procedures and virtual methods on the one hand, and that of join points on the other. In fact, we claim we can seamlessly replace *procedure* and *method* in Fig. 1 with *join point*, in the sense of a semantic action to be executed (we will use the term *join point action* to denote this action). When an advice is bound to a join point, the latter's semantic action consists of the advice execution as well as the original action if it is not omitted, e. g., by an around advice that does not proceed.

From an aspect-oriented point of view, a program looks like in (a) if it is not written in an AOP language: crosscutting concerns are tangled with the application and scattered over it. AOP languages allow to localise these concerns, but current implementations of these languages for the most part *early-bind* advice to join points, sometimes guarded by conditional, so-called *residual* [42] logic. The target code these implementations generate resembles part (b) from the figure. Although conditional logic is generated by the AOP language implementation, it is part of the application code. An implicit dispatch for join points in the target code as in part (c) should be the goal of AOP language implementations.

A logical consequence is to regard every single join point as a locus of late binding, i. e., as a *virtual join point*. An, in this regard, conceptually clean implementation of a run-time environment for aspect-oriented programming languages implicitly represents join points as virtual join point "calls". Each such call is dispatched at run-time and one target is selected according to the current run-time state. The original join point action is, among possibly applicable advice, contained in the set of potential targets. If no advice apply to the join point, there is only one potential target: the default join point action. This is comparable to a virtual method that is not overwritten.

We will now consider how powerful dispatch has to be. In object-oriented programming languages, the standard case is to dispatch a method call only based on the receiver object's type. This can be realised by using a dispatch table. *Multi dispatch* [20], where the receiver and argument types are taken into account, requires extended mapping from multiple types to a method. *Predicate dispatch* [26, 50] is the most general notion, attaching an arbitrary predicate to a method: if it evaluates to true, the method is executed.

When a virtual method table is used, only *one* run-time object can influence dispatch, usually the method call receiver. However, AOP languages allow for richer semantics in pointcuts, and pointcut expressions are usually more complex, so that dispatch is oriented along more than one dimension. In the following, we will briefly discuss which dimensions of dispatch are met in existing AOP languages.

AspectJ [42, 4] provides dynamic pointcut designators `cflow`, `target`, `this` and `args`, which specify the current control flow, dynamic type of receiver, active or argument objects, respectively. Consequently, dispatch has to regard these.

Other AOP implementations like CaesarJ [3, 17], JAsCo [67, 66], Association Aspects [60], Steamloom [33, 34, 9], PROSE [56, 57, 52, 58] or EOS [59] 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.

Even more dimensions are conceivable that hint at the capabilities of upcoming and future AOP languages. If, for example, a pointcut language regards the history of execution [66, 1, 55] or the interconnections of objects on the heap [55], dispatch dimensions come into scope that are laborious to implement with a purely object-oriented target machine. The generalised concept of virtual join point dispatch, when realised at the core of an execution environment, delivers a more powerful basis on which such languages can be built.

3 Delegation-Based AOP

In this section, we will first introduce the delegation-based AOP machine model in its simplest form, i. e., in a purely prototype-based setting. After that, we will show how the purely prototype-based model can be extended to support class-based languages. A brief discussion and summary close this section.

3.1 Prerequisites

The machine model for AOP proposed here is based on the concepts of *prototypes* and *delegation* [46]. The join point model's granularity is that of *messages*, i. e., each message send constitutes a join point. Both method invocations and member accesses are equally modelled as messages sent to receiver objects. It is this feature by which the model facilitates late binding at all join points: the exact locus of late binding is message reception.

In Fig. 2(a), a single object `obj` is shown. It has three slots responding to the messages `foo`, `bar`, and `baz`. The implementation of the message `bar` sends the message `foo` to `self`, i. e., to the very object that received the `bar` message. The parent of `obj`—parent references are represented as arrows—is some object further up the delegation chain of objects.

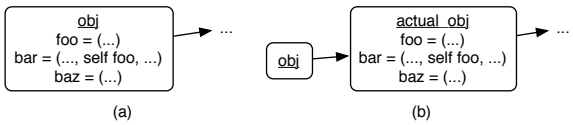

Fig. 2. (a) A single object with an unspecified parent, (b) an object and its proxy.

In the context of the execution model for AOP proposed herein, an object is not referenced directly, but through a proxy, as shown in Fig. 2(b). The proxy, by default, does not understand any particular messages, but transparently delegates all messages sent to it to the object it stands for. In the figure, `obj` is the proxy object by the name of which the actual object `actual_obj` is known.

Technically, the proxy object determines the actual object's identity at all times: objects might be inserted to or removed from the delegation chain, but since the proxy object will always remain up front, references to it will never need to be updated.

Additionally, as calls are delegated up the delegation chain, `self` will always be bound to the proxy object. For example, when `bar` is sent to `obj`, the call is delegated to `actual_obj`, where the message is understood. Its implementation sends `foo` to `self`. The latter, because `bar` was *delegated* to `actual_obj`, is still bound to `obj`.

3.2 Introducing Aspects

We will now turn to showing how the common mechanism of delegation can be used to late-bind advice to join points. Assume there are two aspects `asp_a` and `asp_b`. Both affect different messages in `obj`: `asp_a` adds a *before* advice to `bar` and an *around* advice to `baz`, `asp_b` adds an *after* advice to both `foo` and `bar`. Both aspects are dynamically deployed at different moments in time while the application is running.

Fig. 3 shows the situation after `asp_a` has been deployed. An additional object, named `asp_a_proxy`, has been inserted in the delegation chain between the proxy and the actual object. This so-called *aspect proxy* understands the two messages augmented by the corresponding aspect, namely `bar` and `baz`.

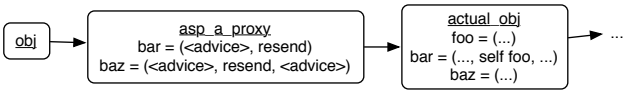

Fig. 3. The aspect `asp_a` has been deployed.

The effect of this delegation chain modification is that all messages sent to the actual object via its proxy are *first* understood by the aspect proxy, bringing about the application of advice. From here on, the aspect proxy acts as a smart reference (hence our usage of the term *proxy* [30]) to the actual object: it performs actions of its own, as well as possibly addressing `actual_obj`. For example, `bar` is understood in `asp_a_proxy`. The aspect proxy's implementation of the message applies advice functionality before it *resends* the message, i. e., passes on the message while `self` remains bound to the original receiver, `obj`. This means that the original implementation of `bar` in `actual_obj`, when it is eventually executed, correctly sends `foo` to `obj`.

Please note that the figures do not make any assumptions as to where advice functionality is actually implemented; it may be given in-place, i. e., in the aspect proxies themselves, or the latter may call other objects to execute advice.

Next, `asp_b` is deployed as well. The resulting situation is shown in Fig. 4. The aspect proxy for `asp_b` has been inserted in the delegation chain between the aspect proxy for `asp_a` and the actual object.

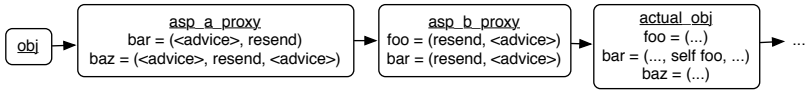

Fig. 4. Both `asp_a` and `asp_b` have been deployed.

The situation after the deployment of `asp_b` is especially interesting with regard to the messages `bar` and `foo`. The former is subject to a *before* and an *after* advice introduced by `asp_a` and `asp_b`, respectively. The use of delegation in the machine model facilitates transparent advice application to `foo`: when `bar`'s original implementation sends `foo` to `self`, the message is routed through the proxy `obj` and both aspect proxies, leading to its interception in `asp_b`.

In the example, the aspect proxy of the last-deployed aspect was inserted immediately before the actual object in the delegation chain, which means that the first-deployed aspect applies first. Different orders of advice application are

straightforward to achieve by reordering aspect proxies in the delegation chain. Aspect precedence can thus easily be dealt with: it basically is a matter of proxy ordering.

The need for a proxy is now apparent. All modifications due to dynamic weaving affect the delegation chain *leading to* the decorated object. Without the proxy, all references to that object would have to be updated upon dynamic aspect deployment. The proxy ensures a unique reference at all times, making delegation chain modifications between itself and the actual object transparent.

The above examples employ *before*, *after* and *around* advice. In the figures, all advice actions are subsumed under `<advice>`. It is obvious that delegation-based AOP easily facilitates all three types of advice in that it treats before and after advice as special cases of around advice.

A crucial part of all message implementations in aspect proxies is the execution of the decorated join point. In delegation-based AOP, this is achieved by *resending* the respective message to the next object in the delegation chain, during which `self` still remains bound to the original message receiver.

3.3 Adding the Thread Dimension

So far, the description of the model has only shown how late binding is facilitated along two dimensions, namely the identity of the receiver of a message send, and the message itself. We will now show how additional dimensions can be supported, and we will use *thread locality* as the first example for this.

Thread locality can be observed in existing AOP implementations in two forms. On the one hand, aspects can be *scoped* to a single given thread, or a number of threads. That is, their advice apply to join points only when the latter occur in the execution of the respective thread(s). This feature is, for example, directly supported in CaesarJ [3] and Steamloom [33]. On the other hand, thread locality may imply that different (advice or residual [42]) functionality must be executed depending on the thread at hand. For example, the AWED language [51] allows for per-thread aspect instantiation. It also is a core requirement for `cflow` residues to be thread-local, i. e., to maintain control flow information *per thread*.

The *current thread* is thus added as a dimension of dispatch at join points. The delegation-based AOP machine model allows for addressing both forms of thread locality in a uniform way. To that end, the `parent` reference of each object is defined to be a *function of the current thread* rather than a static reference. That way, an object's `parent` can be different, depending on the current thread. Essentially, the delegation chain itself becomes a property of the thread.

For illustration, Fig. 5 shows, again, the sample object `actual_obj` and its proxy, `obj`. This time, two aspects `asp_c` and `asp_d` have been deployed. The former introduces a *before* advice to `foo` that only applies in a thread `T1`, the latter introduces a *before* advice to `bar` that applies globally.

In the figure, the dashed line with the annotation "T1" denotes a delegation link that applies in the thread `T1`, while solid lines denote unconditionally effec-

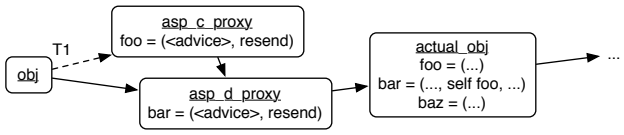

Fig. 5. The aspect `asp_c` is scoped to the thread `T1`, `asp_d` applies globally.

tual links. It can be seen how `asp_c_proxy` delegates to `asp_d_proxy`, effecting the application of `asp_d` in all threads.

3.4 Adding the Control Flow Dimension

Next, we will show how the introduced model mechanisms can be used to support yet another dimension of dispatch, namely the current control flow. This basically models the `cflow` construct known from AspectJ [42].

The sample aspect in this case, `asp_e`, applies a before advice to `foo` *only* if this message is sent in the control flow of an execution of `bar`. In the model, this is achieved using *continuous weaving* [32], i. e., the corresponding aspect proxies are dynamically inserted into and removed from the delegation chain as the control flow in question is entered and left. It is important to note that this has to take place *per thread*: when the control flow is entered in `T1` but not in `T2`, only the delegation chain of `T1` is to be affected.

Consider Fig. 6 for illustration. In part (a), the situation is shown where `asp_e` is deployed but no thread is currently in the control flow of executing `bar`. Still, `asp_e_cw_proxy`—a *continuous weaving proxy* pertaining to `asp_e`—has been inserted in the delegation chain. It serves the purpose to dynamically deploy the actual aspect proxies whenever a thread enters or leaves the respective control flow. Note that the `<activate>` and `<deactivate>` functionality surrounds the `resend` of the control-flow constituting message like an *around* advice.

Fig. 6(b) shows the situation after `bar` has been sent to `obj` in a thread `T1`. For that thread, the delegation chain is different now; sends of the `foo` message are understood in `asp_e_proxy`, where advice functionality is applied. Note that the continuous weaving proxy is *not* in the delegation chain for `T1`, so as to avoid multiple insertions of the aspect proxy due to recursive entries of the control flow.

In Fig. 6(c), another thread, `T2`, has entered the control flow. The continuous weaving proxy has reacted to this by simply adding `T2` to the set of threads for which the "parent function" of `obj` yields the aspect proxy `asp_e_proxy`. That way, advice apply to `foo` in both `T1` and `T2`, but in no other thread.

This approach to handling the actual aspect proxy guarantees that the aspect proxy is inserted into the delegation chain at most once. The model's property of regarding parent references as functions allows for adding and removing particular threads to the set of threads for which the parent function yields the aspect proxy. The aspect proxy is not removed until the last thread leaves the respective control flow. This is taken care of in the continuous weaving proxy.

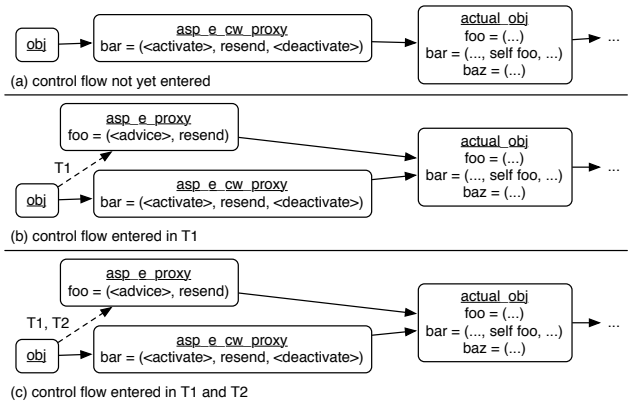

Fig. 6. Dispatch along the control flow dimension through continuous weaving.

It is important to stress that no extra features were introduced to the model in order to support the control flow dimension. A continuous weaving proxy is technically identical to any other aspect proxy, or indeed any other object. The only requirement is the capability to dynamically modify an object's delegation chain, while the delegation mechanism handles message flow.

3.5 Supporting Class-Based Languages

The delegation-based AOP machine model is originally based on *prototypes*. We will now show how the model can easily be extended to support class-based languages while retaining all benefits from the prototype-based version, such as instance-local and thread-local aspect deployment.

It is easy to emulate the class-instance relationship known from class-based languages in a prototype-based setting [64, 12]: any class is represented by an object defining the class behaviour, while any instance of a class, represented by an object whose parent slot points to the class, only carries its *state*. The instantiation of an object is done by cloning a prototype.

In the extended delegation-based AOP model, objects have references to their classes, and the way methods are invoked along these references can be modified by modifying the path to the class. Fig. 7 shows how the basic principle works: every object (`c` in the figure), as seen before, is represented by a proxy that references the actual object (`actual_c`). The actual object contains instance-specific attributes, i. e., member fields. The actual object in turn does not directly reference its class, but it does so via another proxy, the so-called *class proxy* (`proxy_C`) whose purpose will be clarified below. Finally, the *class* is represented by an object (`C`) that defines the messages any instance of the class understands.

If an aspect `asp_f` with a class-wide before advice for the message `C.bar` is inserted, the delegation chain is modified as seen in Fig. 8. An aspect proxy `asp_f_proxy` is inserted in between the class proxy and the class. The proxy

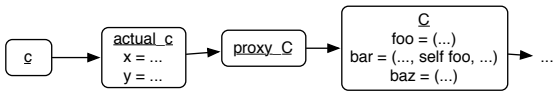

Fig. 7. Objects representing the class `C` and an instance thereof.

understands, exactly in the fashion of the execution model as presented above, the message `bar` and applies advice before resending it.

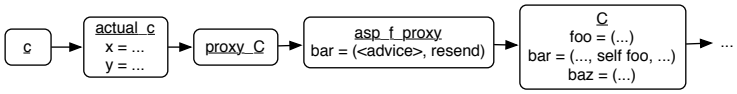

Fig. 8. The aspect `asp_f` introduces a class-wide advice for `C.bar`.

So, the default class proxy is needed because inserting a class-wide aspect without having this proxy would involve changing the parent links of all currently existing instances of the respective class, as well as those of all instances of the class that are created while the aspect is deployed. Hence, the class proxy exists for the same reasons as the default object proxy introduced above.

There is no interference of the mechanisms for class-wide aspects with those for instance-specific decoration. In fact, class-wide and instance-local decorations can be seamlessly combined. The underlying mechanism is always delegation of messages through proxies. In Fig. 9, there are two instances `c1` and `c2` of the class `C`. Both are connected to their corresponding class object via the default class proxy. However, there is another proxy on the delegation path for `c1`. In fact, this proxy object implements the message `bar` to form aspectual behaviour, but this new behaviour takes effect *only* if `bar` is sent to `c1`. In the same way, the message `foo` is affected by a before advice—but this advice applies to *all* instances of `C` because its place in the delegation chain is *after* the class proxy.

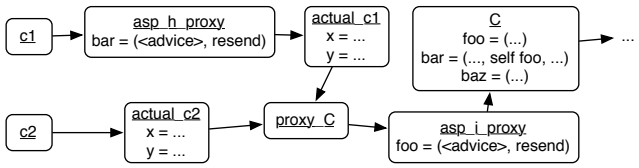

Fig. 9. Two instances of `C`, where one is affected by the aspect `asp_h`, and a class-wide aspect `asp_i`.

Proxies for instance decoration are always well isolated from proxies for class decoration, as the latter are inserted between the class proxy and the class, while

the former are inserted between the decorated instance and the class proxy. Due to this, instance decorations *always* dominate class decorations.

The concepts relating to extended support for dispatch dimensions introduced earlier also apply in this setting: advice can be restricted to particular threads by making the corresponding parent references functions of the thread. This is illustrated in Fig. 10, where `asp_g` applies only in the thread `T1`.

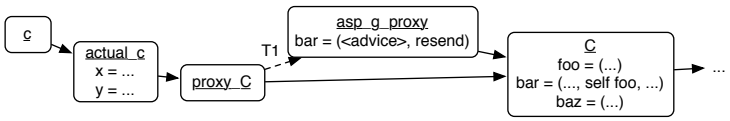

Fig. 10. `asp_g` applies only in the thread `T1`.

3.6 Introductions

The delegation-based AOP machine model does not only support *pointcut-and-advice*-flavoured AOP [48]. We will now show how it easily facilitates *introduction* of fields and methods.

Assume an aspect `asp_int` introducing a field `f` and message `msg` to the class `C`. The situation just after the aspect's deployment is shown in Fig. 11(b) (part (a) shows the situation before deployment). An aspect proxy, `asp_int_proxy`, has been inserted in the usual fashion in between the class proxy for `C` and `C` itself. The proxy understands two messages, namely `msg` and `f`. No fields have been added yet to either `c1` or `c2`. This is done dynamically, as we will see next.

Above, it was mentioned that `asp_int` introduces a *field* `f`. However, the inserted aspect proxy understands a message of that name which is realised as a *method*. The purpose of this method is to facilitate the dynamic on-demand introduction of fields to objects.

Consider what happens when the field `f` of the object `c2` shall be accessed: the message `f` is delegated until it is understood in `asp_int_proxy`. The implementation of `f` inserts an instance-local aspect proxy for `c2` which solely contains the new field `f`, establishing the situation shown in Fig. 11(c).

`f` is now realised as a *field*, which has no method-like functionality to execute and hence does not proceed, like advice implementations. Thus, whenever the message `f` is sent to `c2` in order to access the field, the message is understood in `asp_int_c2` and not delegated further up the delegation chain.

3.7 Discussion and Summary

The machine model for AOP introduced above is based on the well-known concepts of prototypes and delegation, which have been augmented with the additional property that parent references can actually be functions. In fact, the

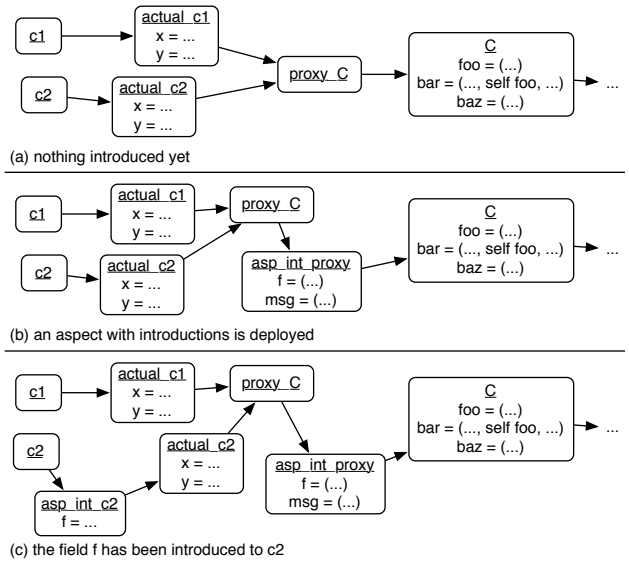

Fig. 11. Introductions in delegation-based AOP.

model allows for such a function to determine its result based on arbitrary parameters, not just the current thread, to realise dispatch along multiple dimensions.

The model, being object-oriented itself, can be used as an execution layer for object-oriented programming languages. As seen in Sec. 3.5, class-based object-oriented languages can easily be supported. We argue that the strengths of the model fully come into play when languages are to be implemented that require extensive use of late binding. Thus, it is especially well suited to support aspect-oriented programming languages.

The model supports the pointcut-and-advice flavour of AOP [48] straightforwardly. Apart from that, the model allows for implementing extended features. Scoping aspect applicability to single instances comes as a natural feature of the model. Yet, other features such as thread-local scoping and per-thread advice—illustrated by the first and second examples, respectively, in Sec. 3.4—are also supported in a unified way: both are done implicitly through parent functions.

Delegation-based AOP also provides very simple mechanisms for realising different aspect precedence strategies. The order of aspect proxies in the delegation chain may depend on several factors, such as the order in which deployment occurs, or explicitly declared precedence. The model also naturally supports dynamic weaving through its reflective capabilities. Proxies can, at all times, be dynamically inserted in and removed from delegation chains.

Set aside the features of pointcut-and-advice AOP, the model also supports *introduction* of fields and methods to existing objects and classes. This is easily achieved simply by exploiting the model's inherent mechanisms. In a nutshell,

the delegation-based AOP machine model represents a uniform approach to implementing AOP, based on some simple yet powerful mechanisms.

A proof-of-concept implementation for the delegation-based AOP machine model has been developed as well. Emphasising elegance and simplicity more than efficiency, the relatively young, dynamic Io programming language [37] was used for this purpose. Regarding an efficient implementation, we refer to existing work on efficiently implementing dynamic languages that was achieved in the course of implementing the Self language [65, 18, 61] and the Strongtalk Smalltalk implementation [63]. In those projects, very efficient compiler technology for dynamic languages has been developed. Adopting their achievements for delegation-based AOP is a core topic of future work.

4 Semantics

We will now introduce the δ [2] calculus, followed by a number of modifications and extensions in order to use it as a formal foundation for our model.

4.1 The δ Calculus

δ is a simple calculus providing a formal foundation for an imperative, object-based system with delegation. It is defined through an operational semantics function $\rightsquigarrow_\delta$, which is a finite mapping of expressions and stores onto pairs of addresses and stores:

$$\rightsquigarrow_\delta : Exp \times Store \mapsto_{fin} Address \times Store$$

A *store* is basically a lookup table which maps addresses to objects, and stores the *self* pointer:

$$Store = (\{self\} \mapsto Address) \cup (Address \mapsto_{fin} Obj)$$

Finally, an object contains a list of addresses, pointing to its parents (δ indeed allows for multiple parent objects), which are each associated with an identifier, as well as a list of method names with their bodies, and are represented as $o \equiv [\![d_1 = \iota_1 \ldots d_k = \iota_k \| m_1 = b_1 \ldots m_n = b_n]\!]$:

$$Obj = (DelegateID \mapsto_{fin} Address) \cup (MethodID \mapsto_{fin} Exp)$$

A number of operations are defined as well which determine the way expressions are constructed, and of which the following are most relevant in the context of this paper:

$$(Addr) \quad \frac{}{\iota, \sigma \rightsquigarrow_\delta \iota, \sigma}$$

$$(Clone) \quad \frac{a, \sigma \rightsquigarrow_\delta \iota, \sigma' \quad \iota' \notin dom(\sigma') \quad \sigma'' = \sigma'[\iota' \mapsto \sigma'(\iota)]}{clone(a), \sigma \rightsquigarrow_\delta \iota', \sigma''}$$

$$(Select) \quad \frac{a, \sigma \rightsquigarrow_\delta \iota, \sigma' \quad Look(\sigma', \iota, m) = \{b\} \quad \sigma'' = \sigma'[self \mapsto \iota] \quad b, \sigma'' \rightsquigarrow_\delta \iota', \sigma''' \quad \sigma'''' = \sigma'''[self \mapsto \sigma(self)]}{a.m, \sigma \rightsquigarrow_\delta \iota', \sigma''''}$$

(*Addr*) is the basic case, where an address evaluates to itself without modifying the store. (*Clone*) performs a copy-by-value of an object's parents and methods, and stores the result at a new address. Finally, (*Select*) models message sending and makes sure *self* is initialised to point to the message receiver. The *Look* function basically looks up the method body associated with m, either in the receiver object itself, or in one of its parents. It is assumed that only one candidate is found.

The delegation semantics are thus incorporated in the *Look* function, but as the latter will be modified to better fit the context of this paper (cf. Sec. 4.2), its original definition is omitted here.

4.2 Modifications and Extensions for Delegation-Based AOP

For δ to be convenient as a formal foundation for delegation-based AOP, a number of adaptations need to be made. First of all, a simplification can be applied, in that it turns out to be sufficient for each object to have maximally one parent instead of n. This is because (*Select*) will exhibit the same behaviour if a message is sent to an object $[\![d_1 = \iota_a, d_2 = \iota_b \| \ldots]\!]$ as if the same message were sent to $[\![d_1 = \iota_a \| \ldots]\!]$ where $\sigma(\iota_a) = [\![d_1 = \iota_b \| \ldots]\!]$. Indeed, in both cases, lookup will check b's methods only after it failed to find a suitable candidate in a. However, in order to allow an object's parent to vary depending on the context (for example the current thread), a function *Del* is introduced, which associates every object with another function. The latter, in turn, determines the object's parent based on the context. Consequently, parents will no longer appear in an object's representation:

$$Del : Address \mapsto_{fin} (Context \mapsto Address)$$
$$o \equiv [\![m_1 = b_1, m_2 = b_2 \ldots]\!]$$

Note that the *Context* domain is not defined in more detail in order to allow it to be used for any information considered applicable in a particular situation. Furthermore, the notation Del_ι will be used from now on as an abbreviation for $Del(\iota)$ and, in case $Del(\iota)$ is a constant function, even for that constant value. For convenience, Del_ι is assumed to be stored together with the actual object in the store at address ι.

Next, (*Clone*) should be updated to make sure objects are automatically associated with a proxy, and can be referenced through this proxy:

$$\frac{\begin{array}{l} a, \sigma \rightsquigarrow_\delta \iota, \sigma' \\ \iota' \notin dom(\sigma') \\ \sigma'' = \sigma'[\iota' \mapsto \sigma'(\iota)] \\ \iota'' \notin dom(\sigma'') \\ \sigma''' = \sigma''[\iota'' \mapsto ([\![\,]\!]; Del_{\iota''}(context) = \iota')] \end{array}}{clone(a), \sigma \rightsquigarrow_\delta \iota'', \sigma'''} \quad (Clone)$$

Note that $Del_{\iota''}$ is set to be a constant function here. This means that the parent of the proxy object will always be the actual object, regardless of context. Furthermore, the proxy object has got no methods of its own. Thus, all messages sent to it are automatically delegated to its parent.

Also note that the semantics of $(Clone)$ as defined here may not be suitable in all cases, for example to create an aspect proxy object which does not need another proxy of its own. For such cases, the old $(Clone)$, or even yet another variant, might be more appropriate. The current version demonstrates what it means for a proxy to be attached to an object, as well as how and when this might be realised.

As stated before, delegation semantics are incorporated in the *Look* function, which is now adapted to take the *Del* function into account. More specifically, it should look for a method m in the object at address ι or any object found by recursively applying the *Del* function, and return its body together with the address of the object where m was eventually encountered:

$$Look(\sigma, \iota, m) = \begin{cases} \{(b, \iota)\} & if\ \sigma(\iota) = [\![\ldots m = b \ldots]\!] \\ Look(\sigma, Del_\iota(context), m) & otherwise \end{cases}$$

Note that $Look(\sigma, \iota, m)$ is undefined if at some point an application of Del_ι is undefined as well. This will happen in case an object has no parent.

At this point, delegation semantics are suitable for delegation-based AOP. The next issue is that there is no *resend* mechanism yet. In order to incorporate this, two new pseudovariables *msg* and *cur* are introduced, which, similarly to $self$, are only relevant during a message send:

$$(Var)$$

$$\begin{array}{l} \overline{self, \sigma \rightsquigarrow_\delta \sigma(self), \sigma} \\ cur, \sigma \rightsquigarrow_\delta \sigma(cur), \sigma \\ msg, \sigma \rightsquigarrow_\delta \sigma(msg), \sigma \end{array}$$

Indeed, a *resend* is only possible within the body of a method, and *msg* and *cur* respectively serve to hold the name of the message currently being handled, and the address of the object where the body of this message was found by the *Look* function. Consequently, $(Select)$ is now modified to correctly initialise these new variables, and the definition of a *store* is updated as well:

$$(Select)$$

$$\frac{\begin{array}{l} a, \sigma \rightsquigarrow_\delta \iota, \sigma' \\ Look(\sigma', \iota, m) = (b, \iota_d) \\ \sigma'' = \sigma'[self \mapsto \iota][msg \mapsto m][cur \mapsto \iota_d] \\ b, \sigma'' \rightsquigarrow_\delta \iota', \sigma''' \\ \sigma'''' = \sigma'''[self \mapsto \sigma(self)][msg \mapsto \sigma(msg)][cur \mapsto \sigma(cur)] \end{array}}{a.m, \sigma \rightsquigarrow_\delta \iota', \sigma''''}$$

$$Store = \begin{array}{l} (\{self\} \mapsto Address) \cup (\{cur\} \mapsto Address) \cup \\ \qquad (\{msg\} \mapsto MethodID) \cup (Address \mapsto_{fin} Obj) \end{array}$$

At this point, (*Resend*) can be modelled to select *msg* on the parent of *cur*, while *self* is not modified, and thus remains bound to the original receiver:

$$\frac{\begin{array}{l}(Resend)\\ Look(\sigma, Del_{cur}(context), msg) = (b, \iota_d)\\ \sigma' = \sigma[cur \mapsto \iota_d]\\ b, \sigma' \rightsquigarrow_\delta \iota', \sigma''\\ \sigma''' = \sigma''[cur \mapsto \sigma(cur)]\end{array}}{resend, \sigma \rightsquigarrow_\delta \iota', \sigma'''}$$

Note that *cur* is updated during (*Resend*). This is necessary to cover the case where the evaluation of the newly found b triggers yet another *resend*.

Finally, a couple of dedicated aspect-oriented operations can be defined. It turns out that deploying an aspect is just a matter of rewiring a couple of parents, while aspect undeployment boils down to resetting this rewiring:

$$\frac{\begin{array}{l}(Deploy\ Aspect)\\ a, \sigma \rightsquigarrow_\delta \iota, \sigma'\\ asp, \sigma' \rightsquigarrow_\delta \iota', \sigma''\\ \sigma''' = \sigma''[Del_{\iota'}(context) = Del_\iota]\\ \sigma'''' = \sigma'''[Del_\iota(context) = \iota']\end{array}}{deploy(asp, a), \sigma \rightsquigarrow_\delta \iota, \sigma''''} \qquad \frac{\begin{array}{l}(Undeploy\ Aspect)\\ a, \sigma \rightsquigarrow_\delta \iota, \sigma'\\ asp, \sigma' \rightsquigarrow_\delta \iota', \sigma''\\ \sigma''' = \sigma''[Del_\iota(context) = Del_{\iota'}]\end{array}}{undeploy(asp, a), \sigma \rightsquigarrow_\delta \iota, \sigma'''}$$

4.3 Example

As an example, consider the scenario shown in Figs. 2(b) and 3 from Sec. 3. We start out with an object $obj = \sigma(\iota_{obj}) = [\![\,]\!]$ and $Del_{\iota_{obj}} = \iota_{actual\_obj}$ where $\sigma(\iota_{actual\_obj}) = [\![foo = \ldots, bar = [\ldots, self\ foo, \ldots], baz = \ldots]\!]$. Thus, *obj* is a proxy object with its parent pointing to *actual_obj*.

Next, *aspA* is deployed using (*Deploy Aspect*):

$$deploy(\iota_{aspA}, \iota_{obj}), \sigma \rightsquigarrow_\delta \iota_{obj}, \sigma'$$

The $Del_{\iota_{obj}}$ function is now set to always evaluate to the constant value ι_{aspA}, but this need not necessarily be the case. The slightly more advanced situation where *aspA* is applied locally to thread T_1 (cf. Sec. 3.3) is easily covered by a minor change to the (*Deploy Aspect*) operation, where $Del_{\iota_{obj}}$ is set to the following instead (t is the current thread):

$$Del_{\iota_{obj}}(t) = \begin{cases} \iota_{aspA}\ if\ t = T_1 \\ Udf \quad otherwise \end{cases}$$

Of course, in case $t \neq T_1$, the result might just as well be yet another object, rather than undefined. The latter models the case where *obj* has no parent.

5 Related Work

The discussion of related work is done in two parts. First, we will focus on work that also supports the notion of join points as loci of late binding. We will then turn to presenting implementations or implementation ideas that exploit mechanisms resembling an actual late-binding approach as presented in the preceding sections.

5.1 Join Points as Loci of Late Binding

Join points as loci of late binding have been alluded to in numerous publications presenting formalisms for aspect-oriented programming. A number of these approaches regard join points as *events* [68, 15, 25, 29, 28] to which advice essentially *react*. While this can be regarded as a form of late binding, the notions of join points used in the aforementioned publications still differ from our idea in that they assume that *certain* join points are selectively *activated* as events during a weaving step [25], or that additional conditional logic is executed whenever such an event is signalled to determine whether advice are actually applicable [68]. The application is under observation, it is being monitored by some entity pertaining to AOP infrastructure. Conversely, our model regards *all* potential join points as being "active" and thus implicitly as loci of late binding at all times. Moreover, additional conditionals are not required in delegation-based AOP because late binding is done *implicitly* through the appropriate insertion of proxies in the delegation chain.

In the AOSD-Europe project, a generic meta-model for aspect-oriented programming languages has been developed [14]. It explicitly regards join points as points where advice functionality may be late-bound. The model comes with a prototype implementation in the form of an interpreter, which checks for advice applicability at all join points it encounters. This corresponds to an "eager" checking for the applicability of advice. Application of advice is thus less implicit than in delegation-based AOP.

Some formalisms explicitly address *dispatch mechanisms* to model join points and advice application at them [53, 45, 38]. The Fred language [53] combines concepts from object-oriented and aspect-oriented programming as well as predicate dispatch [26]. This approach is close to delegation-based AOP regarding its derivation. Still, it requires the definition of conditions for dispatch at application level instead of applying dispatch implicitly, like our model.

Lämmel introduces *method call interception* (MCI) as a fundamental language mechanism [45]. MCI allows for superimposing method calls with additional functionality. The MCI model is, however, restricted to method calls only and does not aim at representing a general model for AOP.

The calculus of untyped aspect-oriented programs presented by Jagadeesan et al. [38] is very closely related to the delegation-based AOP model presented in this paper. It models all advice applying at a join point as ordered units of behaviour, each of which is essentially an *around* advice. Such an advice unit closely resembles an aspect proxy in delegation-based AOP. Advice are also

implicitly applied at join points. Only method calls, which can be expressed using message sends, are considered as join points in the calculus.

The *parameterised aspect calculus* [19] regards *each* reduction step as a potential join point. Still, the semantics consults a pointcut language element at all reduction steps, leading to eager explicit checking for advice applicability like observed above for the AOSD-Europe meta-model.

The *common aspect semantics base* (CASB) [23] regards every instruction as a potential join point, applying a two-staged function at all instructions. The first function determines whether the current *instruction* may be subject to decoration with advice. If so, the second function is applied to check whether the present *dynamic state* calls for applying advice at the join point at hand. An instruction is hence treated like an AspectJ join point shadow. The view on a running application therefore closely resembles the one found in AspectJ.

In *Pluggable AOP* [44], aspect language mechanisms are modelled as mixins that transform interpreter base mechanisms. Said mixins are represented in the form of proxy objects, and composition with the base interpreter functionality is achieved through proxy insertion in delegation chains that are part of the interpreter's logic. The similarities with the machine model for AOP presented in this paper are of a technical nature. Pluggable AOP augments interpreter base mechanisms by means of proxies and delegation. Conversely, delegation-based AOP has proxies and delegation *as the interpreter's core mechanisms*. In other words, Pluggable AOP transforms the interpreter, whilst delegation-based AOP transforms application structures subject to execution by a never-changing interpreter.

Ossher [54] proposes to represent application objects as "constellation[s] of a number of fragments" that each contribute part of an object's functionality. Fragments delegate to each other in case a piece of desired functionality is not implemented by one. Crosscutting concerns can be dynamically woven in and out by adding fragments to, or removing them from, the delegation chain. His proposal is a suggestion for research directions for virtual machine support for concern composition. The machine model presented herein obviously matches with these ideas.

5.2 Related Implementations

There are several actual AOP language implementations that use techniques related to late binding at join points. None of them is as radical as the model presented in Sec. 3, but certain resemblances exist.

Envelopes as in *envelope-based weaving* [16] wrap potential join point shadows in methods introduced at load-time. They closely resemble virtual join points but are limited in that they basically just map virtual join points to virtual methods. Some AOP implementations utilise a less consequent form of envelopes to realise dynamic weaving. AspectWerkz [5, 11] does not replace *all* potential join point shadows with envelopes, but those that are, at class loading time, known to be in the scope of aspects that may be put to use during run-time. Each join point shadow is replaced with a call to a method in a dedicated so-called *join point*

class. Dynamic weaving is achieved by replacing said methods using HotSwap [24, 41]. AspectWerkz' approach is described in more detail in [13]. JAsCo [67, 66], in its run-time weaver [39], follows a similar approach.

One variant of PROSE [57] decorates *each* join point shadow with advice dispatch logic, effectively realising a powerful dispatch mechanism. However, the approach brought about severe performance penalties, as the chosen implementation strategy was nothing like virtual method dispatch. In fact, advice dispatch logic was implemented as an unconditional callback into the AOP framework of PROSE, leading to the execution of costly functionality at all join points.

Implementing AOP languages using *proxies* is an approach chosen by numerous AOP frameworks, of which Spring AOP [40, 62] is one of the most popular. Frameworks like Spring AOP create proxies that replace the original application objects and implement the same interface as the latter, but apply advice in their implementations of the respective methods. Proxy-based AOP is technically close to the delegation-based AOP model for yet some more reasons. Aspect precedence is easily expressed by ordering proxies appropriately. Also, proxies can be applied—introduced and withdrawn—dynamically, allowing for dynamic weaving. The main difference between proxy-based and delegation-based AOP lies in the level at which the approaches are realised: proxy-based AOP implementations operate at *application level*. That is, all AOP-related operations are part of the running application, imposing significant performance penalties on join points where advice functionality applies [33]. Conversely, delegation-based AOP is intended to be realised at the level of the run-time environment, promising significantly better performance.

The *composition filters* [6] approach is related to proxies in that filters are applied to messages. Filters may impose additional functionality on message evaluation, thereby effecting advice. Execution of such functionality may also depend on conditions specified in filters. Composition filters relate to delegation-based AOP in the same way proxies do: filtering is specified at language instead of machine level.

Finally, there are implementations that do not affect application *code* as such, but that manipulate meta-level entities to let aspect-related constructs take effect. Systems falling in this category are AspectS [36] and context-oriented programming (COP) [21]. Their relation to delegation-based AOP is apparent: both do modify system-internal dispatch data structures, such as virtual method tables or method dictionaries, to augment functionality at join points. The main difference to delegation-based AOP lies in that they do not explicitly regard *all* join points as loci of late binding—this characteristic is introduced by installing aspects (in AspectS) or activating layers (in COP).

6 Summary and Future Work

Based on the notion of join points as loci of late binding, we have presented a machine model for the implementation of aspect-oriented programming languages called *delegation-based AOP*. The model not only facilitates the implementation

of the pointcut-and-advice AOP flavour, but also that of numerous other AOP features, such as aspect scoping (to threads and instances) or introductions. The model is simple and exploits few simple basic concepts—delegation, prototypes, parent reference functions—to achieve all of its goals.

Future work will focus on several issues. An implementation of the model is to be developed in the form of a virtual machine for a high-level aspect-oriented programming language. The machine will support some standard bytecode set (e. g., Java or Smalltalk), but will moreover offer dedicated bytecode instructions supporting the core aspect-oriented features of the machine model. Existing aspect-oriented programming languages are to be mapped to it by means of compilers that target the machine's instruction set. To achieve good performance, existing work on providing efficient run-time environments for dynamic languages is going to be used as a foundation.

Acknowledgements

The authors are grateful for Christoph Bockisch's comments on early versions of this paper. He and Mira Mezini have also contributed to the ideas of this paper during numerous discussions. The authors also thank Klaus Ostermann, Bart Du Bois, Dirk Janssens and Serge Demeyer for their worthwhile suggestions.

The authors thank Joseph Osborn, who has contributed an implementation of delegation-based AOP in Io, elements from which have been used to derive the implementation presented in this paper.

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. Christopher Anderson and Sophia Drossopoulou. δ - an imperative object based calculus with delegation. In *Proc. USE'02*, Malaga, 2002.
3. I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of caesarj. *Transactions on AOSD*, LNCS 3880, 2006.
4. AspectJ Home Page. `http://www.eclipse.org/aspectj/`.
5. AspectWerkz Home Page. `http://aspectwerkz.codehaus.org/`.
6. L. Bergmans and M. Akşit. Principles and Design Rationale of Composition Filters. In [27].
7. C. Bockisch et al. Adapting VM Techniques for Seamless Aspect Support. In *Proc. OOPSLA 2006*. ACM Press, 2006.
8. C. Bockisch, M. Haupt, and M. Mezini. Dynamic Virtual Join Point Dispatch. Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT '06), 2006.
9. C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proc. AOSD 2004*. ACM Press, 2004.
10. C. Bockisch, S. Kanthak, M. Haupt, M. Arnold, and M. Mezini. Efficient Control Flow Quantification. In *Proc. OOPSLA 2006*. ACM Press, 2006.
11. J. Bonér. What Are the Key Issues for Commercial AOP Use: how Does AspectWerkz Address Them? In *Proc. AOSD 2004*, pages 5–6. ACM Press, 2004.

12. A. Borning. Classes versus Prototypes in Object-Oriented Languages. In J. Noble, A. Taivalsaari, and I. Moore, editors, *Prototype-Based Programming. Concepts, Languages and Applications*. Springer, 1999.
13. J. Brichau et al. Report Describing Survey of Aspect Languages and Models. Technical Report AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, Vrije Universiteit Brussel, 17 May 2005 2005. `http://www.aosd-europe.net/deliverables/d12.pdf`.
14. J. Brichau et al. An Initial Metamodel for Aspect-Oriented Programming Languages. Technical Report AOSD-Europe Deliverable D39, AOSD-Europe-VUB-12, Vrije Universiteit Brussel, 27 February 2006 2006. `http://www.aosd-europe.net/deliverables/d39.pdf`.
15. G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μABC: A Minimal Aspect Calculus. In *Proc. CONCUR'04*, pages 209–224. Springer, 2004.
16. C. Bockisch and M. Haupt and M. Mezini and R. Mitschke. Envelope-based Weaving for Faster Aspect Compilers. In *Proc. NetObjectDays 2005*. GI, 2005.
17. CaesarJ Home Page. `http://caesarj.org/`.
18. C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford University, 1992.
19. C. Clifton, G. Leavens, and M. Wand. Parameterized Aspect Calculus: a Core Calculus for the Direct Study of Aspect-Oriented Languages. `ftp://ftp.ccs.neu.edu/pub/people/wand/papers/clw-03.pdf`, 2003.
20. C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proc. OOPSLA'00*, pages 130–145. ACM Press, 2000.
21. P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *Dynamic Languages Symposium (DLS) '05, co-organised with OOPSLA'05*. ACM Press, 2005.
22. T. Dinkelaker et al. Inventory of Aspect-Oriented Execution Models. Technical Report AOSD-Europe Deliverable D40, AOSD-Europe-TUD-4, Darmstadt University of Technology, 28 February 2006 2006. `http://www.aosd-europe.net/deliverables/d40.pdf`.
23. S. D. Djoko, R. Douence, P. Fradet, and D. Le Botlan. CASB: Common Aspect Semantics Base. Technical Report AOSD-Europe Deliverable D41, AOSD-Europe-INRIA-7, INRIA, France, 10 February 2006 2006.
24. M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution, Proceedings (at OOPSLA 2001)*, 2001.
25. R. Douence, O. Motelet, and M. Südholt. A Formal Definition of Crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.
26. E. Ernst, C. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *Proc. ECOOP'98*, pages 186–211. Springer, 1998.
27. R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.
28. R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical Report 01.12, RIACS, May 2001 2001.
29. 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.
30. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

31. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
32. S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving. In *Proc. AOSD 2004*. ACM Press, 2004.
33. M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Software Technology Group, Darmstadt University of Technology, 2006.
34. 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.
35. E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proc. AOSD 2004*. ACM Press, 2004.
36. 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.
37. Io Home Page. `http://www.iolanguage.com/`.
38. R. Jagadeesan, A. Jeffrey, and J. Riely. A Calculus of Untyped Aspect-Oriented Programs. In *Proc. ECOOP'03*. Springer, 2003.
39. JAsCo Home Page. `http://ssel.vub.ac.be/jasco/`.
40. R. Johnson and J. Hoeller. *Expert One-on-One J2EE Development without EJB*. Wiley, 2004.
41. Java Platform Debugger Architecture Home Page. `http://java.sun.com/j2se/1.4.1/docs/guide/jpda/index.html`.
42. 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.
43. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
44. S. Kojarski and D. H. Lorenz. Pluggable aop: Designing aspect mechanisms for third-party composition. In *Proc. OOPSLA '05*, pages 247–263. ACM Press, 2005.
45. R. Lämmel. A semantical approach to method-call interception. In *Proc. AOSD'02*, pages 41–55. ACM Press, 2002.
46. H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. OOPSLA 1986*, pages 214–223. ACM Press, 1986.
47. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
48. H. Masuhara and G. Kiczales. Modeling Crosscutting Aspect-Oriented Mechanisms. In *Proc. ECOOP 2003*, 2003.
49. 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.
50. T. Millstein. Practical predicate dispatch. In *Proc. OOPSLA'04*. ACM Press, 2004.
51. L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed aop using awed. In *Proc. AOSD'06*, pages 51–62. ACM Press, 2006.
52. A. Nicoara and G. Alonso. Dynamic AOP with PROSE. `http://www.iks.inf.ethz.ch/publications/publications/files/PROSE-ASMEA05.pdf`.

53. D. Orleans. Incremental programming with extensible decisions. In *Proc. AOSD'02*, pages 56–64. ACM Press, 2002.
54. H. Ossher. A direction for research on virtual machine support for concern composition. In *Proc. Workshop VMIL '07*. ACM Press, 2007.
55. 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.
56. A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In G. Kiczales, editor, *Proc. AOSD 2002*. ACM Press, 2002.
57. A. Popovici, T. Gross, and G. Alonso. Just-in-Time Aspects. In *Proc. AOSD 2003*. ACM Press, 2003.
58. PROSE Home Page. `http://prose.ethz.ch`.
59. 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. ACM Press, 2003.
60. K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proc. AOSD'04*, pages 16–25, 2004.
61. Self Home Page. `http://research.sun.com/self/`.
62. Spring AOP (from the Spring reference documentation). `http://www.springframework.org/docs/reference/aop.html`.
63. Strongtalk Home Page. `http://www.strongtalk.org/`.
64. D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4(3), 1991.
65. D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227–241. ACM Press, 1987.
66. W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful Aspects in JAsCo. In *Proc. SC'05*, volume LNCS 3628, pages 167–181. Springer, 2005.
67. W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in jasco. In *Proc. AOSD'05*, pages 75–86. ACM Press, 2005.
68. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.