Are Pointcuts a First-Class Language Feature?

Stephan Herrmann*
Technische Universität Berlin
stephan@cs.tu-berlin.de

ABSTRACT

This paper challenges the notion "pointcut" and its role in defining the essence of aspect-oriented programming languages. We present a conceptual experiment of explaining AOP without using the notion "pointcut" demonstrating a large subset of AOP for which simpler and better-explored notions suffice. By the remaining delta to existing aspect languages we narrow down for which precise concept the introduction of a new notion is useful. We discuss why that concept is indeed fundamental and present a model of our understanding of pointcuts as first-class program elements. One of the key properties of the proposed model is its improved type safe compositionality.

1. DO WE KNOW WHAT POINTCUTS ARE?

At the conceptual level the most quoted definition of Aspect Oriented Programming (AOP) uses the terms "quantification" and "obliviousness" to describe the relationship between an aspect and the base program it adapts [9]. This definition serves as an umbrella for a number of programming languages which realize the promises of AOP in quite different ways. The definition does not preclude which language features a programming language must exhibit in order to be called aspect oriented.

Mainstream AOP languages de-facto agree on a notion that has been coined in the context of AspectJ: *pointcut*.[1] Definitions of this notion mainly focus on the intentional level by saying *pointcuts* are a means for capturing sets of points in the execution flow of a program. But what actually *are* pointcuts? Can a language without a concept of pointcuts still be considered an AOP language? If we need pointcuts, what would be the most appropriate theory and calculus behind them?

These are the kind of questions that this paper approaches. The questions have been discussed in depth in the team developing the programming language ObjectTeams/Java (OT/J)[18] and its tools [19]. Our goals were to analyze existing pointcut languages in order to determine the key features that, according to the state of the art, a pointcut language must support. After such analysis the best-of-the-breed sub-language was to be integrated into OT/J in order to complete the design of this language.

Much to our surprise the longer this discussion lasted, the less we knew what should be considered the commonly agreed-upon state of the art with respect to pointcuts. At the beginning of this discussion we tried really hard to even avoid the notion of pointcuts (for lack of a precise understanding) and tried to define all essentials of AOP using the notion "join point" as the only fundamental concept regarding the desired "quantification". While this experiment of defining AOP without "pointcuts" was quite successful to a certain degree, eventually it became more and more clear to us, what actually could be the role of the notion "POINTCUT" in the foundation of AOP.

Our investigation results in two seemingly opposite suggestions:

1. A large subset of aspect oriented programming can be described completely without the notion of pointcuts.
2. For pointcuts to be an integral part of a programming language it is probably a good idea to make pointcuts first-class elements of programming.

While discussing the issues at hand we pursued the fundamental goal of defining the notion of POINTCUTS together with a *calculus* for composing complex structures from elements. Such a calculus should provide composition operators for POINTCUTS that are type-safe in a stricter sense than what is realized in current AOP languages. By this we mean, that any POINTCUT specification that passes the type checker should have a sound interpretation. Some counter-examples will be given in Section 6.

We would like to point out that this paper does not try to define a formal semantics for AOP but it separates existing AOP mechanisms into simpler mechanisms which are already well-understood. The focus lies on reducing the concept of "pointcuts" to simpler concepts, or concepts already existing in standard object-oriented languages. This reduction, we hope, will improve formal and informal understanding of existing and future AOP languages.

Throughout the discussion we will relate individual concepts to previous publications on the given topic. An overall comparison to related work, however, is postponed towards the end of this paper.

*This work has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

[1]Throughout this paper we will typeset *pointcut* referring to the AspectJ notion, while we propose a related but restricted notion: POINTCUT.

2. FOUR STEPS TOWARDS POINTCUTS

This section describes a set of fundamental concepts and mechanism which may be used to approximate AOP without using the notion "pointcut". This discussion serves two purposes: First, we are looking for an improved understanding of the elementary concepts behind AOP. In our understanding the concept "*pointcut*" is too ambitious for this purpose, its interpretations are so broad that this concept can hardly be considered elementary. Second, by attempting to define AOP without *pointcuts* we will isolate a specific concept that cannot be captured with the conceptual tools used up-to that point (Sect. 3). Section 4 will give an interpretation that motivates why we consider that isolated concept to be indeed fundamental. At that point we will agree to use the notion "POINTCUT" and give a model of how this concept can be folded back into object-oriented programming (Sect. 5).

2.1 Method Call Interception

At the outset of our discussion we adopted the definition by Ralf Lämmel who claimed *the* fundamentally new concept in AOP to be *method call interception* (MCI) [16]. His paper demonstrates how the dynamic semantics of an AOP language can be defined by adding nothing but MCI to a given imperative object-oriented programming language.

Examples

Each AOP language features some mechanism for binding advice/aspect methods to join points. The mechanism of all these aspect bindings can be interpreted as MCI. Xu et al [22] suggest a similar reduction of AOP to an existing dispatch mechanism. They chose *implicit invocation* as a foundation, which may be taken as a hint that the dispatching can be controlled by an entity that's neither part of the base nor of the aspect proper, but some kind of *mediator*.

Throughout this paper we will present small code examples in the syntax of ObjectTeams/Java (OT/J) [14], so Figure 1 shows an example of declaring MCI in OT/J, here an *after callin binding*, saying that `aspectMethod` should be invoked after each execution of `baseMethod`.

Interpretation

Technically MCI is closer to reflection based mechanisms whereas AOP is usually explained in terms of code weaving. Since semantically both mechanisms are equivalent either model is suitable for defining the semantics of AOP. In fact, we believe that MCI is also an appropriate model for *teaching* AOP, maybe more so than weaving, because MCI allows for an intuition of what the actual source code will produce, whereas weaving requires the indirection of explaining another (transformed) version of the program that nobody will ever see. Along the same lines Lämmel argues that MCI can be *realized* by either program transformation or (behavioral) reflection ([16]). However, for a *change of attitude* Lämmel envisions that MCI will be considered fundamental just like dynamic binding (which can also be realized by program transformation or reflection). Once MCI will have been granted a similar standing as dynamic binding, aspect-oriented thinking will be much more natural than understanding programs only by the transformation of weaving.

```
void aspectMethod() <- after void baseMethod();
```

Figure 1: Method call interception in OT/J

Aside from weaving and reflection a third model has been proposed: event based communication (see, e.g., [7, 22]). In this paper we will frequently speak of events during the execution of a program because we feel that this well supports the intuition of MCI. However, for the context of AOP we consider weaving, reflection and event based communication pretty much as different (interchangeable) views of the same ideas.

In fact, MCI fully realizes the concept of "obliviousness" in that sense, that a base program can be developed in ignorance towards any aspects. Any control flows from the base code into the aspect code are defined by the aspect only.

Desirable capabilities

When binding methods of the base program to aspect methods data flows need special attention, because neither method may be designed explicitly for this binding, so signatures may exhibit arbitrary mismatches.

For this purpose, OT/J supports so called *parameter mappings*, by which parameters can be re-mapped between base and aspect methods. For `before` and `after` bindings such mappings are one-way, but in `replace` bindings parameter mappings must be reversible in order to map a base-call (comparable to `proceed`), which uses the signature of the aspect method, back to the actual base method.

Parameter mappings are comparable to AspectJ's concept of *context exposure*. For the discussion at hand method signatures only play a minor role.

2.2 Join Points

The most obvious restriction in the AOP-as-MCI view regards the operations that can actually be intercepted. Clearly, the focus on method calls does not suffice for most aspect languages. This is where we need to introduce the next fundamental notion: the notion of "join points" as points in (the static structure of) the program.[2] With this new notion we should generalize the interception mechanism to cover all kinds of join points, thus speaking of *join point interception* (JPI).

Join points are elements within a program that can be referred to either by name or by more complex specifications. Technically speaking, those program elements correspond to *nodes* in the abstract syntax tree (AST) of a program. Reasoning about AST nodes from within the program means to support a specific kind of *structural reflection* (as opposed to the behavioral reflection that can realize JPI).

[2]We'd like to mention that the word "point" in this discussion should not be interpreted in its strict mathematic way, which would imply a concept with no extension and no other properties other than its location in space. Join points definitely have both an extension (so before and after a point are indeed distinguishable) and some relevant properties (so it is relevant to replace a point by something else).

At a closer look the actual reflexive representation of a program opens at least two new degrees of freedom:

1. What should be the actual granularity of this program representation?
2. Should the representation be the plain AST or should a resolved graph representation (ASG) be chosen?

Question 2 will be discussed in the next section. Motivation for discussing the *granularity* (1) can, e.g., be drawn from the evolution of AOP in general as well as of the language AspectJ. In the beginning one might have uniformly thought of join points as relating directly to method calls where method bodies would largely be seen as black boxes. Meanwhile even statement level join points are being discussed[3]. More specifically, AspectJ started with a quite limited number of kinds of join points. In 2000, e.g., AspectJ featured these join points: calls, receptions[4] and executions of methods and constructors, setting and getting a field as well as execution of exception handlers. The current version of the AspectJ programming guide[5] mentions 11 kinds of join points. Research still suggests new kinds of join points as, e.g., in [13]. It is difficult to see, at what point this evolution will terminate.

From analyzing what types of nodes are used to express Java programs in source code *and* in byte code we can identify between 17 and 20 kinds of join points which are suitable for aspect binding. Here are two examples of join point kinds that are not standard in today's aspect languages, which could indeed by useful in real world programming:

- A `cast` join point could, e.g., be used to implement things like *custom conversions* in C# [1]. This join point kind has also been suggested in [4].
- A `dereference` join point could intercept the "." operator by which the receiver of a method call or field access is evaluated/dereferenced. This kind of join points could very well support different styles of proxies and/or caching in a most elegant style. We are not aware of previous publications about dereference join points.

Note, that both these join point kinds are even sub-statement granularity: they capture individual *expressions* or *operators*. As a counter-example, all kinds of branching or jump instructions are not suitable for interception.

We have reasons to believe that within the given range a complete set of join point kinds can be determined.

[3]Statement level join points would, e.g., be useful for instrumenting programs for the purpose of measuring test coverage (cf. [21].

[4]receptions have later disappeared from the language

[5]Unfortunately, the language documentation makes no mentioning of the language version it defines, so quoting "the current version of AspectJ" simply means AspectJ as defined by the "Programming Guide" as I downloaded it today from [3].

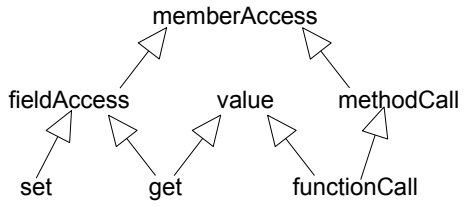

Figure 2: A multi-hierarchy for join point kinds.

Once the set of node types in this representation is defined it is also crucial to define a good *taxonomy* providing abstractions over related types of nodes. While such taxonomy is common place in compiler implementations, we suggest to extend existing taxonomies towards *multi-hierarchies* in order to allow abstractions following different criteria.

Using a multi-hierarchy the following abstractions could coexist in the same model (see Figure 2): A 'field access event' abstracts over read and write access to a given field or a set of fields. A 'value producing event' abstracts, e.g., over reading a field and calling a function.[6] The following table illustrates matching for this example using the fictitious join point kinds `fieldAccess` and `value`:

	`fieldAccess(f)`	`value(T)`
`this.f = new T();`	match	—
`return this.f;`	match	match
`return this.getF();`	—	match

2.3 Join Point Queries

Join point interception as described in the previous sections can be regarded as supporting the desired "obliviousness" property of AOP. The next step towards aspect-oriented happiness [7] requires to add a concept for "quantification". The question is, given the above definition of *join points*, how do we obtain the desired *sets* of join points, ie., how is quantification actually specified? In this question we agree with Eichberg et al [8], who suggest to define sets of join points using *functional queries*. Given a graph representation of the program any query language capable of traversing graphs should be suitable. From the analysis of existing *pointcut* languages, the following three elements seem to be a good foundation for a query language for join points:

1. A *scope* defines where to search by selecting a set of elements to search within.
2. A *kind* defines which type of nodes is searched.
3. An optional *constraint* may further restrict the set of nodes found so far. A constraint is a predicate over a potential join point (just like the `where` clause in many query languages).

[6]Note that the latter abstraction is a built-in feature of the language Eiffel. Meyer calls this the principle of uniform access[17]. In languages that do not realize this principle in their syntax, a multi-hierarchy may still establish the desired abstraction a-posteriori.

[7]Cf. [17].

In contrast to the predominant use of ASTs, a resolved ASG (question 2 in the previous section) allows to define join point sets not only over containment relationships but over any associations in the language's meta model. Several authors feel inclined to consider the call graph as the spanning structure for computing sets of join points.[8] Clearly call graphs require the abstract syntax structure to be resolved. A second use of resolved information is in constraints that may be applied as filters to a previously defined set of join points. Many useful constraints require references in the AST to be resolved.

Examples

The static part of AspectJ's *pointcut* language is an example for a join point query language, although AspectJ is better explained by a the concept of *matching* join points rather than querying. Logic meta programming [12, 20] allows to define sets of join points by prolog predicates over the reflexive program representation. In [8] the (functional) query language *XQuery* is used basically for the same purpose.

Interpretation

It is clear that the definition of sets of join points requires capabilities that go beyond explicit enumeration. Matching as supported by AspectJ yields a more concise notation whereas systematic query languages (be they logic or functional in nature) are better suited as a foundation of AOP. Clearly, a good query language contains as a subset the expressiveness of pattern matching.

As long as query languages are applied only statically, ie., at compile time, this model is also practical because the effort for evaluating static queries should be more or less in the same order of magnitude as the compilation proper. The performance of the final program is obviously not effected by the choice of query language.

2.4 Event Filtering

The sets of join points defined by queries as discussed in the previous section can be interpreted as triggers to which aspects can be bound using the mechanism of join point interception (JPI). Given this framework for quantification (join point queries) and obliviousness (join point interception) several researchers have felt the desire for dynamically filtering the set of join point invocations in order to apply JPI only under given conditions which have to be evaluated at run-time.

Please don't confuse these dynamic filters with the constraints that may contribute to the queries defined above. Constraints as part of a join point query reason about static program properties, whereas event filtering is based on run-time values.

Examples

Obviously the Composition Filters [5] approach is based centrally on the idea of filtering event flows. Also AspectJ's `if` *pointcut* provides a basic means for the same purpose. A more modular mechanism for dynamically enabling a set of aspect bindings is defined by *aspect activation* (sometimes referred to as dynamic aspect deployment). Aspect activation may affect an aspect class or an instance, it may be controlled implicitly or explicitly, as a block construct or in a purely imperative style, and it may affect different scopes of a running system. ObjectTeams/Java supports a variety of means for controlling aspect activation (see Paragraph 5 in [14]).

[8] E.g., the `pcflow` pseudo-*pointcut* requires a call graph analysis (see [8] for an initial discussion on this topic).

```
void aspectMethod() <- after void baseMethod()
   when (condition);
```

Figure 3: Method binding guard in OT/J

The most explicit and versatile way of event filtering at run-time is given by OT/J's guard predicates [15]. Figure 3 shows the binding of an aspect method in OT/J which is guarded by a predicate (keyword `when`).

Interpretation

Since JPI introduces control flows which are invisible in the program, the question arises where these control flows themselves could be controlled if they should happen only conditionally. A method invocation that is not explicit in the program cannot easily be guarded by a condition. Instead of forcing the programmer to implement such conditionals within the imperative aspect code, the afore mentioned approaches provide means to express conditionals at locations closely connected to the JPI mechanism. Yet, the guard concept in OT/J [15] is more powerful than the others as it may control aspect activation and *instantiation* and it may do so at four different levels of granularity, thus enhancing modularity of the activation aspect of aspects.

We favor the idea of describing JPI in terms of events. In this understanding conditional JPI means to provide mechanisms for filtering call events at run-time. Strictly speaking, event filtering is not necessary for explaining AOP. As mentioned before, any event filter could be hand-coded into the respective aspect methods. However, guards as featured by OT/J are a fairly lightweight language feature which in return provide significant improvements in modularity and readability. No new theory is needed to embed guards into the foundation of AOP, since the triple *join point*, *filter*, *aspect method* actually realizes the well-explored paradigm of *event-condition-action* systems like, e.g., statecharts (see the discussion in [15]).

Desirable capabilities

For languages supporting aspect *instances* it is desirable to select whether filtering should occur before or after lookup of an aspect instance.[9] For complex aspects modularity can be enhanced if filters can be applied to program elements at different levels of granularity. OT/J supports to attach a guard predicate to (a) a method binding, (b) an aspect method, (c) an aspect class ("role") and (d) a compound module of aspect classes ("team").

[9] This distinction not only determines the set of values available for the guard, but allows to selectively suppress on-demand instantiation of an aspect.

3. WHAT IS MISSING?

The four concepts *join points*, *join point interception*, *join point queries* and *event filtering* allow us to explain almost everything that can be expressed in today's AOP languages. Note that so far we saw no need to introduce the notion "pointcut" for our explanations. Of course the AspectJ view of our discussion would have used the notion "*pointcut*" throughout. We have two reasons for being reluctant towards this notion: (1) In all cases where "*pointcut*" is only a synonym for "set of join points" a new notion is simply not needed. (2) *If* we introduce a new notion we would like to see a precise definition first.

Having come so far without saying "pointcut", what is the actual delta between AOP languages with and without a "pointcut" concept? We will agree to use the notion "POINTCUT" if a concept of AOP can be found that cannot be expressed by the above notions. This agreement pre-assumes that the yet-to-be-defined concept will actually turn out to be a cohesive concept with well-defined properties. Looking at AspectJ we see exactly one construct that cannot be described in the afore mentioned set of concepts: `cflow`. Other, newer concepts come to mind, too: stateful aspects [6] and tracematches [2].

3.1 Classifying dynamic aspect binding

Before focussing on the above three concepts, let's first check whether this list is actually complete. Stateful aspects and tracematches should be seen as representatives for a group of related approaches. Looking at AspectJ we find six kinds of *pointcuts* that describe some kind of dynamism: `cflow`, `cflowbelow`, `if`, `this`, `target` and `args`. The `if` construct has already been discussed as directly supporting what we call event filtering. An analysis of `this`, `target` and `args` is blurred by the fact, that each of these constructs realizes two quite unrelated concepts: context exposure[10] and matching according to the dynamic type of certain run-time values. For simplification we suggest that the matching part of these *pointcuts* be rephrased using an appropriate `if` *pointcut* (in this sense `target(T o)` is equivalent to `if(o instanceof T)`). Thus `this`, `target` and `args` do not introduce a new concept that has not already been covered by concepts in the previous section.

Indeed the `cflow` construct (and of course `cflowbelow`) remains as the only AspectJ *pointcut* that cannot be expressed simply in those terms defined in the previous section.

3.2 Commonality of advanced concepts

Let's compare the three mechanisms not yet covered by our set of concepts: `cflow`, stateful aspects and tracematches. We find one striking similarity: All these mechanisms reason about more than one event (instance). Usually several events are considered which are generated by different join points. These events occur at different points in time. Whether or not the execution of aspect code is fired by a given event depends on other events occurring at other points in time.

[10]Context exposure is not a major focus of this paper. This topic has been touched briefly as "parameter mapping" in section 2.1.

Care should be taken not to confuse static quantification with the dynamic, temporal multi-event property of POINTCUTS. For each join point in a static set (as defined by a static join point query) always *one* event suffices to trigger the aspect code. The same even holds for the (dynamic) event filters discussed in Sect. 2.4: one trigger is enough. Thus we see three levels of dynamism:

1. No dynamism in static queries (quantification).
2. Run-time snapshots for evaluating filters (guards).
3. Multi-trigger POINTCUTS whose evaluation evolves over time.

We use the word "POINTCUT" exclusively in situations where multiple event instances decide about triggering or not triggering some given aspect code. We decide to do so because this understanding of "POINTCUT" cannot be reduced to simpler fundamental concepts. At a first sight one might be tempted to select one of the given approaches as the foundation trying to define the others in the terms of the selected approach. We have two reasons for not doing so:

1. The stateful aspects and tracematches approaches, though being closely related in spirit, bear one incompatibility which forbids direct translation from one notation to the other: the semantics of the automata implicitly underlying both approaches differ in (at least) one decisive point: Each transition of a stateful aspect may actually trigger advice. By this tool all states of the automaton are potentially observable and it is ensured that the automaton at each point in time has exactly one state. By contrast, a tracematch may trigger only one advice after the whole pattern has been detected. Before that point the state of the automaton is not observable, even 'worse': it may be in several states potentially simultaneously when matching overlapping sequences.

   Thus we see no way to declare one approach as fundamental and map all other approaches to it.

2. Although common properties of existing multi-trigger POINTCUTS can be identified, we could imagine other approaches appearing in the future, which also consume several triggers into one POINTCUT but with different semantics of this combination.[11]

4. POINTCUTS ARE REVERSE METHODS

Do the above considerations provide good reasons for introducing a new concept into the theory of programming languages? After we narrowed down the concept to be defined to those situations that require to consider several run-time events, we actually do not see any existing concept (outside the field of AOP) that could easily serve for explaining what we want to explain.

However, we can pin-point a well-established concept that actually performs the reverse operation of what POINTCUTS shall do: *methods*! We could actually think of POINTCUTS

[11]A wide field for such innovation would perhaps be given when detecting patterns of concurrent behavior!

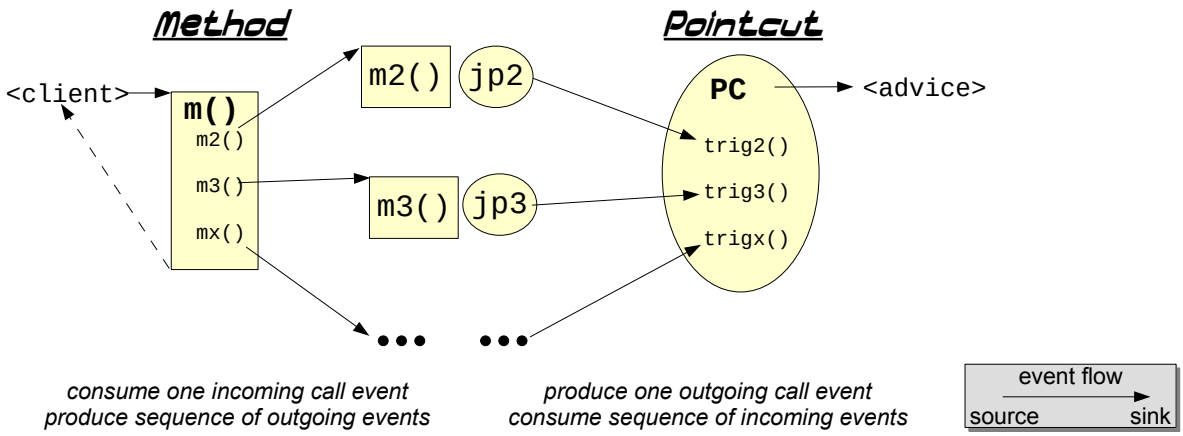

Figure 4: Interpreting a Pointcut as a reverse method.

as reverse methods. If we can demonstrate such an anti-symmetry between methods and POINTCUTS we consider this as sufficient motivation for the concept of "POINTCUTS", because if such a thing as a "reverse method" exists, it is certainly (a) a fundamental concept (b) not directly supported by traditional languages.

What would it mean to define POINTCUTS as reverse methods? Looking at the intention behind both of them we could define methods as program elements responsible for producing the behavior of the program. Instead of *producing* behavior, POINTCUTS are intended for *detecting* actually performed behavior. A more precise definition of this anti-symmetry looks at the fan-in and fan-out of run-time control flows:

A method is always invoked by *exactly one* caller, i.e., any method invocation consumes exactly one call event. Conversely, a POINTCUT may consume *any number* of call events before actually doing anything.

After consuming its input events, the POINTCUT may eventually produce exactly one output event. Advice may be bound to this event in order to perform aspect behavior. The purpose of methods, on the other hand, lies in their capability to call any number of (other) methods, which should be interpreted as producing a number of output events.

Furthermore, a method knows those servers to which it sends method call events, ie., the method knows about connected event sinks. The method is however ignorant towards its (potential) clients — the event sources connected. By contrast, a POINTCUT knows about connected event sources only (the elementary events being observed) and is ignorant towards the sink which will eventually consume its output event. Figure 4 illustrates this anti-symmetry.

These observations are well in line with previous work in which POINTCUTS have been described, e.g., as monitors[7].

5. POINTCUTS AS FIRST-CLASS ENTITIES

After we have recognized "POINTCUTS" as being a fundamental new concept, we now look for a model that may provide a conceptual realization of what we have described as "reverse methods". Rather than defining this model from scratch we use existing constructs for this conceptual realization.

As a suitable model for POINTCUTS we propose a special use of classes. Such a class contains one method for each input event. The input events are connected to these methods by join point interception, which can be established as a static connection. These input methods may perform arbitrary local action. The methods may read any system state that is visible to them but they should usually update only internal state of the POINTCUT.

In addition to these input methods a POINTCUT has one special method called `fire`. This method is always empty. Based on the local state of a POINTCUT any of the input methods may call `fire` in order to signal the complete detection of the behavior specified by this POINTCUT. As an alternative the method `fire` may be directly bound to a given input event but guarded by a filter that will enable the firing only in a specific state of the POINTCUT.

Aspect code may be connected to this fire event by another level of method call interception.

Examples

In addition to the examples from existing approaches (cflow, stateful aspects and tracematches), one could easily think of a number of specific patterns that could be captured as POINTCUT classes within a POINTCUT library.

The simplest example we could think of is a `once` POINTCUT, which for the execution of a system fires only once (useful for all kinds of initializations). Here the POINTCUT simply maintains a flag whether it has already fired.

```
1  // emulates cflow(call (B1.m1())) && call(B2.m2())
2  class MyCFlow extends Pointcut
3  {
4      // dispatch logic:
5      boolean enabled = false;
6      callin void trigger1() {
7          boolean oldVal = enabled;
8          enabled = true;
9          base.trigger1(); // the base-call
10         enabled = oldVal;
11     }
12
13     // connecting incoming triggers:
14     trigger1    <- replace B1.m1;
15     fire        <- replace B2.m2
16         when (enabled);
17 }
18
19 // binding the above pointcut to an aspect method
20 class MyRole
21 {
22     void roleMethod() { ... }
23
24     // aspect binding using a callin binding:
25     roleMethod <- before MyCFlow.fire;
26 }
```

Figure 5: Emulating cflow using a Pointcut class

Figure 5 gives a sketch of how cflow can indeed be implemented using a POINTCUT class. Lines 5–11 implement the required dispatch code. Only while executing the first base method via the base-call in line 9, the flag `enabled` is set to `true`. Line 14 binds this piece of infrastructure to the first base method, here `B1.m1`. The binding in line 15 is guarded by the predicate `(enabled)`. Line 25 finally binds the POINTCUT to the aspect method `roleMethod`.

If arguments of the first method call should be preserved for exposure to the aspect method (the 'wormhole effect'), no explicit stack is needed because the call stack will keep a copy of these values within the stack frame of each invocation of `trigger1`.

Syntax

Please note, that the model presented here is meant to be a conceptual foundation. A language implementation may or may not use this model directly. In the conclusion of this paper we briefly mention two development tasks needed to yield a productive programming language: (1) Supporting specialized syntax for standard POINTCUTS like `cflow`. (2) Optimizing translation of standard POINTCUTS into efficient code. While this might appear as an unnecessary detour as compared to existing aspect languages, we are convinced that a language supporting generalized concepts can much easier incorporate specialized support for specific constructs than vice versa. Thus the model of POINTCUTS as classes should be considered as the point of reference by which a universal, extensible language design is made possible.

It is beyond the scope of this paper to systematically discuss the question of *instances* of POINTCUT classes. However, since POINTCUT classes are now in fact aspect classes, one should expect existing concepts for aspect instantiation to provide sufficient solutions for instantiation of POINTCUT classes, too.

6. TOWARDS A CALCULUS FOR POINTCUTS

One of the most debatable properties of AspectJ relates to the way complex *pointcuts* can be constructed from simpler ones. By using only one syntactic category for join point sets, static join point filters and dynamic conditions the syntax allows to define *pointcuts* whose semantics cannot directly be derived from the language definition. Consider as an example the *pointcut* `if(true)`. This *pointcut* lacks a set of join points and only defines a (tautological) run-time condition. The intuitive semantics of this *pointcut* is "always". We could find no text describing how to convert "always" into an enumerable set of triggers. By contrast we claim, that a *pointcut* without an explicit set of join points is not well-defined. A second example which produces the same problem in quite a different manner is the *pointcut* `!call(foo(..))`. By negating a single trigger we get the set "all possible events but one".[12] Again "all possible events" lacks a sound definition.

Thirdly, the operators `&&` and `||` are problematic mostly because they allow many expressions that define empty sets of join points. For a large number of such nonsensical *pointcuts* it is trivial to detect the inherent contradiction. E.g., the *pointcut* `call(foo) && set(bar)` can never match any join points regardless of its arguments `foo` and `bar`, simply because it defines the intersection of disjoint sets (join point kinds). We argue that a type system for an extension of a statically typed language as Java should preserve the property that structurally invalid programs be detected as violations against the typing rules.[13] In our model `call(foo) && set(bar)` would be syntactically illegal because each join point can only have one kind, whereas this *pointcut* tries to intersect the kinds `call` and `set`.

We propose to solve this problem by a clear distinction into the following syntactic categories:

- Each aspect binding requires a join point set defined by a scope, a kind and optional constraints (Sect. 2.3).
- An aspect binding links a trigger defined by a join point query to an aspect method, establishing the join point interception (JPI) mechanism for the involved program elements (Sect. 2.1).
- An aspect guard defines run-time conditions for actually triggering the aspect depending on run-time values (Sect. 2.4).

For a valid aspect binding the following elements are obligatory: scope, join point kind, binding kind (before, after,

[12]Note, that this is different from "all calls except those to foo". By negating a join point the expression is actually changed into a join point constraint, which is then applied to the empty *pointcut*, which *probably* means "always".

[13]Indeed the AspectJ compiler can detect a number of problems in *pointcut* definitions. The problem is, users may write many linguistically correct *pointcuts* which at a closer look are *probably* not what he or she wanted to express. Not only for the name of the corresponding compiler option (`Xlint`) one feels to be thrown back to the time of `lint` which was motivated by C's property of allowing many programs that should be considered dangerous.

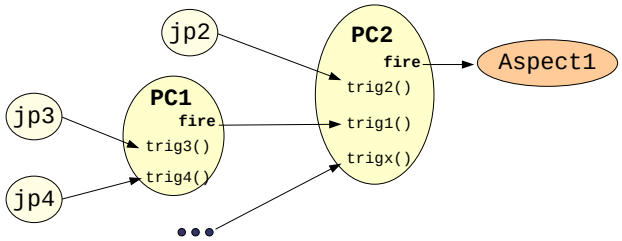

Figure 6: A cascade of Pointcuts.

replace) and aspect method. Static constraints (as part of the query) and run-time guards are optional.

An AOP language may also support set operations and/or functional operators over several join point queries, provided the join point kinds and their signatures are compatible, but composition of different categories is syntactically prohibited.

When extending this model to include POINTCUTS as defined in the previous sections the central questions is how composition of POINTCUTS can be integrated into this calculus. In previous approaches we find two extreme answers: In AspectJ *pointcuts* can freely be combined with each other and with any element of a join point specification. The tracematches approach defines exactly two levels of pointcuts: elementary (AspectJ) *pointcuts* and tracematches, defining histories of elementary *pointcuts*. In this approach tracematches simply cannot be composed further. We see how this two level approach avoids additional complexity, but if POINTCUTS are to be seen as first-class things, we should not content without a universal calculus for POINTCUT composition.

The interesting point about realizing POINTCUTS as classes lies in the fact that compositionality is given for free: By defining POINTCUTS merely as a disciplined use of aspect classes, existing combination concepts suffice to compose POINTCUTS from other POINTCUTS. Recall that the purpose of a POINTCUT class is to collect input events (via inbound JPI) and produce one output event defined as a call to the predefine method `fire`. By explicitly representing the firing of a POINTCUT in the imperative part of the language, this event automatically becomes a potential join point that can be intercepted by aspect bindings at the next level. Until now we only saw how an actual aspect method is bound to the firing of a POINTCUT. But if an aspect class may intercept the fire event, so can any other POINTCUT class (by virtue of being an aspect class). This way it is possible to construct cascades of POINTCUTS, each POINTCUT consuming events from the lower level POINTCUTS or join points, which is illustrated in Figure 6.

The ability to create cascades of POINTCUTS is very well in line with the interpretation of POINTCUTS as reverse methods. The effect of executing a method is the execution of those methods that are mentioned in method calls within the method body thus creating a call graph of arbitrary depth. Conversely, a POINTCUT can be triggered by the firing of other POINTCUTS which are mentioned within the POINTCUT definition thus creating a trigger graph of arbitrary depth. A POINTCUT with only static join points as triggers is a leaf in this graph.

This anti-symmetry suggests that POINTCUTS could even be recursive, meaning that the firing of a POINTCUT can be consumed by the very same POINTCUT again. Perhaps POINTCUT recursion is just a purely academic exercise, because any effect constructed by such POINTCUT recursion can also be implemented within the methods of the POINTCUT. On the other hand, perhaps a history POINTCUT could use this mechanism for re-initializing itself after each firing.

We leave the question whether recursive POINTCUTS are useful to be answered by future practical application. What matters more is the fact that for the composition of POINTCUTS there's absolutely no need to define a new calculus. By defining the firing of a POINTCUT as the call to the special method `fire` the new construct is folded back into the base language. Thus any reflection about method calls automatically applies to the firing of POINTCUTS, too. In addition to direct cascades of POINTCUTS this also entails the option to quantify over POINTCUTS. Within a join point query fire methods of POINTCUTS can be matched just like any other method. This allows to define aspects that, e.g., monitor all Pointcut firing of another aspect. Please note, that this may be quite different from observing the execution of aspect methods. Firstly, an individual POINTCUT may fire without invoking any aspect method (yet). Secondly, for a given aspect method it may be interesting to observe which individual POINTCUT actually caused its execution.

7. RELATED WORK

Throughout the paper we have shown to what degree previous publications support the individual points of our argumentation. The most relevant sources for inspiration were: Lämmel's semantics based on method call interception [16], Eichberg's suggestion to use a query language (XQuery) for defining sets of join points [8]. The work on event based AOP (EAOP) provided the metaphors *event* and *monitor* [7] as well as the definition of stateful aspects [6]. Our concept of POINTCUTS as reverse methods could be seen as a refinement of the aspect-as-monitor view. Also, the abc-group stimulated our thoughts by their work on tracematches [2].

In early stages of our research, logic meta programming [12] was a favored candidate for our join point language. In our perception of aspect-oriented research this was the first pointcut language based on an established calculus. Later, Ostermann et al have shown the maximum expressiveness of a pointcut language based on logic meta programming (ALPHA) [20]. Practical applicability of their meta-circular interpreter still remains to be shown. Our efforts in this paper followed quite a different goal: Instead of adding a most expressive second language (Prolog) and a set of rich data models we tried to boil down the issue of aspect binding to the simplest possible concepts and mechanisms. While ALPHA provides a uniform framework for all kinds of pointcuts, we see advantage in a model, that clearly separates parts introducing different degrees of dynamism. In this sense, much like in AspectJ, ALPHA does not support to

distinguish between binding to a join point versus binding to a pointcut. Such distinction is relevant because in our model all bindings to join points can be resolved statically, while a POINTCUT requires runtime computation and additional dispatch. In neither AspectJ nor ALPHA there is an easily identifiable language subset that can be resolved statically. Also type checking across sub-languages poses new problems.

A marriage of both concepts could, however, be envisioned when allowing a POINTCUT class in our model to query a prolog engine whether or not to fire at a given input event. This would restrict the ALPHA approach by forcing the programmer to make all shadows explicit. Advantages of such potential marriage have not been investigated, yet.

Our critique of AspectJ has been discussed throughout the paper. In essence we identified the lack of different syntactical categories for fundamentally different elements and the lack of a taxonomy of join point kinds. As a result the mechanisms for *pointcut* composition appear as ad-hoc solutions. Clearly that language defines a landmark and newer languages must demonstrate that they don't fall behind the capabilities of AspectJ. Our interest in discussing AspectJ was in making explicit those individual concepts that are in one way or other blurred and blended in AspectJ.

As mentioned in the introduction, this paper does not try to give a formal model of AOP. Rather than giving a complete specification of a particular problem we try to give a full picture of how behavioral cross-cutting can be founded on simple, well-understood concepts. The set of elementary concepts should provide a suitable basis for aspect languages that are both practical and supportive of static reasoning.

8. CONCLUSION AND FUTURE WORK

The model presented in this paper separates the following ingredients of aspect binding:

1. join points
2. sets of join points as defined by queries over the program's ASG. These queries can again be decomposed into parts *scope*, *kind* and *constraints*.
3. join point interception as the mechanisms for binding aspect methods to events in the base program
4. POINTCUT classes as a composition mechanism for multi-event triggers.

Conditional aspect bindings can be expressed by attaching filters to the JPI mechanism.

All POINTCUTS that require multiple trigger events for firing reason about sequences of events over time. In an imperative host language sequences of events are produced by sequences of statements. It seems most natural to also use the imperative part of a language also for *detecting* sequences of events. Aspects that require this capability must be stateful aspects in order to store information about how much of the sequence has already been detected.

When saying that the imperative part of a language is suitable for detecting sequences of events, we ignored the fact, that an imperative implementation of such monitors is much more verbose than more specialized notations as in [7] or [2]. We currently work on applying model transformation techniques to easily provide simpler syntax for certain POINTCUT patterns like cflow and history POINTCUTS.

Of course naive implementations of history POINTCUTS also cannot reach the efficiency of tracematches as realized by the abc compiler, let alone compare to efficient implementations of the `cflow` construct. The remaining challenge is a challenge of compiler optimization in the following sense: commonly used POINTCUT structures should be specified by pre-defined POINTCUT classes shipped as libraries that are bundled with the compiler. While at client level such POINTCUT classes behave exactly as specified, the compiler is free at translating the pre-defined POINTCUTS into arbitrarily optimized code.

The model of POINTCUTS as aspect classes finally provides a desirable *closedness of language* as it has been requested in [11]: in our model it is possible to have POINTCUTS quantifying over other POINTCUTS.

9. REFERENCES

[1] C# language specification, 2. edition. Standard ECMA-334, 2002.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 345–364. ACM Press, 2005.

[3] The AspectJ Team. The AspectJ^{TM} Programming Guide. http://www.eclipse.org/aspectj/doc/released/progguide.

[4] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In Peri Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 87–98. ACM Press, March 2005.

[5] Lodewijk Bergmans and Mehmet Akşit. Principles and design rationale of composition filters. In Filman et al. [10], pages 63–95.

[6] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, March 2004.

[7] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and*

Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192, pages 170–186. Springer-Verlag, September 2001.

[8] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, LNCS, pages 366–382, Taipei, Taiwan, November 2004. Springer-Verlag Heidelberg.

[9] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

[10] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[11] Kasper Graversen and Kasper Østerbye. Aspects of aspects — a framework for discussion. In *European Interactive Workshop on Aspects in Software*, 2004.

[12] Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 60–69. ACM Press, March 2003.

[13] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In Gary T. Leavens, Curtis Clifton, and Ralf Lämmel, editors, *Foundations of Aspect-Oriented Languages*, March 2005.

[14] S. Herrmann and C. Hundt. ObjectTeams/Java Language Definition version 0.8 (OTJLD). http://www.ObjectTeams.org/def/0.8/, 2002–2005.

[15] Stephan Herrmann, Christine Hundt, Katharina Mehner, and Jan Wloka. Using guard predicates for generalized control of aspect instantiation and activation. In *Dynamic Aspects Workshop (DAW'05), at AOSD 2005*, Chicago, 2005.

[16] Ralf Lämmel. A semantical approach to method-call interception. In *Proc. AOSD'02*, pages 41–55, Enschede, Netherlands, 2002. ACM Press.

[17] Bertrand Meyer. *Object oriented software construction*. Prentice Hall International, New York, second edition, 1997.

[18] Object Teams home page. http://www.ObjectTeams.org.

[19] Object Teams Development Tooling download page. http://www.ObjectTeams.org/distrib/otdt.html.

[20] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proc. ECOOP'05*, Glasgow, UK, July 2005. Springer Verlag, Berlin.

[21] H. Rajan and K. Sullivan. Generalizing aop for aspect-oriented testing. Technical Report CS-2004-30, Department of Computer Science, University of Virginia, September 2004.

[22] Jia Xu, Hridesh Rajan, and Kevin Sullivan. Aspect reasoning by reduction to implicit invocation. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 31–36, March 2004.