A Classification Framework for Pointcut Languages in Runtime Monitoring

Karl Klose and Klaus Ostermann

University of Aarhus, Denmark
{klose,ko}@cs.au.dk

Abstract. In runtime monitoring and aspect-oriented programming, the execution of a program is monitored to check whether a property – formulated in a pointcut language – holds at some point in the execution of the program. Pointcut languages differ significantly in their expressiveness and the amount of information they utilize about the state of the program's execution, and the relation between different pointcut languages is often not clear. We propose a formal framework that provides the common abstractions of these languages and identifies the points of variability, such that pointcut languages can be compared and classified with regard to their expressiveness and cost. Besides its usage as a common frame of reference for pointcut languages, our framework also gives a precise model of the design space of pointcut languages and can hence help to design future pointcut languages in a more principled way.

1 Introduction

There are many different languages that are used to describe conditions on the state or structure of a running program. These conditions are used to react on specific events or states in the software system. Examples are pointcut languages in aspect-oriented programming [9], break-point conditions in debugging systems [15], or specification languages in runtime verification and security enforcement (e.g., [10]). We will use the terminology from aspect-oriented programming and use the term *pointcut languages* to subsume all of these languages in the subsequent. Depending on the application domain, the program may be halted or additional functionality can be executed (so called *advice*) if a pointcut matches the current point in execution.

In contrast to general purpose programming languages, pointcut languages differ significantly in two important dimensions. First, the expressive power of the languages ranges from simple forms of pattern matching to Turing-complete languages. Second the model of the program execution available in the pointcut language is quite different between systems. Examples for the different flavors of pointcut languages range from simple line numbers used in debuggers via sophisticated domain-specific languages such as the AspectJ [8] pointcut language to the usage of various general computation models, such as regular expressions or context-free grammars over callstacks or execution history [17], or Datalog/Prolog/XQuery queries over some representation of the program execution [1, 14] or the static program structure [7, 5, 4].

The aim of this work is to bring order into the huge and so far quite heterogenous design space of pointcut languages. This heterogeneity is multi-dimensional, and it includes the kinds of information the pointcut language operates on, and the granularity and expressiveness of the language on this information. This makes comparisons between pointcut languages and principled design of new pointcut languages quite difficult, which is an important task, because the differences between the expressiveness of the language and the richness of the execution model have significant impact on the performance of the resulting system. Using a formal model we can identify groups of pointcut languages which have similar runtime and memory performance and can make use of the same implementation and optimization techniques. Finally, a formalization allows to prove whether or not two pointcut languages are equal, and what the differences are, if they are not.

At the moment, the lack of a formalization makes analyses very ad-hoc and difficult. The aim of this work is to improve the current, undesirable situation by the following contributions:

- We give precise formal definitions of pointcut-related terminology as mathematical structures based on the base language's semantics.
- We present a multi-dimensional decomposition of the pointcut language design space and discuss both the significance of the dimensions and classifications of existing languages with regard to these dimensions. In particular we make a clear distinction between the richness of the underlying data model and the expressiveness of the pointcut language itself. This distinction is often not made or left implicit when comparing pointcut languages, leading to confusion about how to compare expressiveness at all.
- We give a methodology on how to compare pointcut languages, based on the formal definitions.
- We discuss how our framework can be used a basis for the formal description of static analysis and program transformations for aspect oriented programs.

The remainder of this paper is structured as follows: First we introduce our notion of joinpoint abstractions and models (Sec. 2). We give examples for models and present a small case study showing two different pointcut models. Next we define pointcuts and pointcut languages (Sec. 3). Building on these prerequisites, we show how pointcut languages can be compared in Sec. 4. In Sec. 5 we discuss language implementation issues in the terms of our framework. Finally, we discuss related work and conclude (Sec. 6 and 7).

2 Semantics and Joinpoint Models

Our first step towards a formal framework for pointcut languages is to define the data model upon which joinpoints and pointcuts can be defined. In the following we introduce small-step semantics as the base notation for the semantics of the underlying *base language* and show how properties of execution traces in this semantics can be described.

2.1 The Semantics of the Base Language

There are several alternative formalism to model the semantics of programming languages, such as *denotational semantics*, *big-step operational semantics*, and *small-step operational* semantics. Of these semantics frameworks, small-step operational semantics is the most suitable framework to talk about pointcuts, since it has a precise and direct notion of a *computation step*, which allows to talk about traces, execution order, etc. Hence in the following we assume that the base language is defined by a small-step operational semantics.

We will model the small-step semantics of our base languages by a binary relation $\longrightarrow$ on a set Σ of states: $\longrightarrow \subset \Sigma \times \Sigma$. A state $\sigma \in \Sigma$ can often be split into several parts, such as environment, expression, heap etc., but for our purposes it is sufficient to assume that the state contains all relevant information that is necessary to compute the respective next computation step.

In some situations it is desirable to change the usual small-step semantics in such a way that information relevant for pointcuts can be extracted more easily. For example, in a Featherweight Java [6] trace, it is not easy to determine the lexical origin of an expression that is about to be reduced, or to see when a method call ends. It is possible to extract this information from a full execution trace, but it is easier to modify the semantics such that these kinds of information are directly available, e.g., by adding labels to expressions in the first example or introducing a "return" expression in the second example. We will see an example for this situation later on.

2.2 Joinpoint Models

Pointcut languages usually do not refer to the state of evaluation directly, because it contains both too much and too little information: While, say, the exact shape of the heap may be irrelevant for a pointcut language, it may need other information which is not included in the current state, such as information about states in the history of the computation. For this reason, pointcut languages are usually based on *joinpoint models*, which contain the information that is necessary to evaluate pointcuts. This information may be an abstraction of the evaluation state, static program information, and even information unrelated to the evaluation, or a mixture thereof.

To this end, we define joinpoints to be abstractions over the current state[1]. States that are of no interest to the joinpoint model (such as entering a `for` loop in AspectJ) are mapped to a special value ϵ.

[1] In terms of Masuhara et al., we use the point-in-time model for joinpoints in this work, because it is more flexible than the region-in-time model [11]

Definition 1 (Joinpoints and Joinpoint Abstraction). *A joinpoint abstraction (JPA)* $\mathbf{J} = (J, \alpha_J)$ *over* Σ *is a set* J *of joinpoints together with a mapping* $\alpha_J : \Sigma \rightarrow J \cup \{\epsilon\}$*, from states to the corresponding abstraction.*

Using a joinpoint abstraction, a concrete trace can be turned into an abstract trace. The actual information that is available to a pointcut, however, is a kind of summary of the abstract trace, which we call *joinpoint model*:

Definition 2 (Joinpoint Model). *A* joinpoint model $\mathbf{M} = (M, T_M)$ *over a JPA* (J, α_J) *is a set* M *of model values together with a* joinpoint transfer function*:* $T_M : M \times J \rightarrow M$*.*

The joinpoint transfer function defines how the joinpoint model value for a state σ is calculated using the previous model value and the state abstraction $\alpha(\sigma)$, so eventually the current model value is a kind of fold over the abstract trace. This means that an initial value has to be provided to calculate the model value corresponding to the first state of an evaluation.

2.3 Example: A simple OO language

In this section we show how the joinpoint abstraction and the joinpoint models of two simple pointcut languages can be expressed in terms of our framework. The joinpoint models of these two pointcut languages we consider are similar to the joinpoint models found in the AOP languages AspectJ and Alpha [14].

We focus on a minimal OO language in the style of Featherweight Java (FJ) [6], a simple calculus that models a minimal core of Java and is defined in terms of a small-step semantics. FJ features classes that may contain fields and methods, and can inherit declarations from another class.

We give the syntax of expressions $\mathcal{E}$ in this language by defining its set of redexes ($\mathcal{R}$) and evaluation contexts ($E[\cdot]$):

$$\begin{aligned} \mathcal{R} &::= v.f \mid v.m(\bar{v}) \mid \texttt{return}(v) \\ E &::= [\cdot] \mid \texttt{new}\ C(\bar{v}, E, \bar{e}) \mid E.f \mid E.m(\bar{e}) \mid v.m(\bar{v}, E, \bar{e}) \mid E; e \mid \texttt{return}(E) \end{aligned}$$

where v is a value, e is an expression, C is the name of a class, f the name of a field, m the name of a method, and $\bar{e}$ is a list of expressions.[2] The basic terms are field accesses ($v.f$), method calls $v.m(\bar{v})$ and method returns $\texttt{return}(v)$. Values are objects of the form $\texttt{new}\ C(\bar{v})$, where the arguments to the constructor are in the same order as the fields of the class.

This definition differs from the original FJ definition in two regards: First, we use an evaluation context [18] to conveniently identify the redex in an expression. Second, we introduce a new `return` redex, which is used to identify the end of a method call. This is an example for a modification of the operational semantics as discussed at the end of Sec. 2.1.

[2] We write $\bar{x}$ for a list $x_1 \cdot \ldots \cdot x_n$, $[]$ for the empty list and $x \cdot \bar{x}$ for list construction.

We will identify the set $\mathcal{E}$ of all expressions in our language with contexts from E with a redex or a value placed in the hole $[\cdot]$ and write $E[r]$ for the expression that has context E and redex r. Following [6], the states of the semantics have the form $\Sigma = P \times \mathcal{E}$, where P is a representation of the program that contains the class declarations. The reduction rules are identical to that of FJ, except that there is one new reduction rule $E[\texttt{return}(v)] \longrightarrow E[v]$, which discards a `return` wrapper once the wrapped expression has been reduced to a value.

An AspectJ-style Joinpoint Model. The joinpoint model in this section is meant to capture the core of the model of AspectJ: the current joinpoint and the callstack abstraction, as required for the `cflow` pointcuts. The first step is to define a joinpoint abstraction as a basis for the joinpoint model. The states which are of interest for a callstack-based joinpoint model are *method calls*, i.e., redexes of the form $v.m(\bar{v})$, where v is a value and $\bar{v}$ is a list of values, and *returning from method calls*, i.e., redexes of the form $\texttt{return}(v)$, where v is a value. In AspectJ, it is also possible to react to field reads, hence we could add redexes of the form $v.f$, too, but we want to illustrate the case how to deal with redexes that are not of interest to pointcuts, hence we assume that in this pointcut language field reads can not be addressed by pointcuts. In AspectJ, the joinpoint model does not include receiver and argument values but only their types[3]. To create the joinpoint abstraction we have thus to replace the receiver and argument values in the method call redex by their respective types. Since values in FJ have the form $\texttt{new}\ C(\bar{v})$ we can give a simple value-type mapping by $\text{type}(\texttt{new}\ C(\bar{v})) = C$. Similar to AspectJ we do not treat the `return` joinpoint as a "joinpoint event", i. e., as a entity directly addressable by a construct of the pointcut language. Instead, we only need them to maintain the callstack model. Thus we can safely ignore the return value in our joinpoint model.

Based on these considerations, joinpoint abstractions in the callstack based model are of the form $J_{\text{AJ}} = \{C.m(\bar{C}), \texttt{return}\}$, where C ranges over class names. The abstraction mapping α_{AJ} simply removes the unnecessary information from the states in the semantics:

$$\alpha_{\text{AJ}}(\sigma) = \begin{cases} \text{type}(v).m(\bar{C})) & \text{if } \sigma = (p, E[v.m(\bar{v})]), C_i = \text{type}(v_i) \\ \texttt{return} & \text{if } \sigma = (p, E[\texttt{return}(v)]) \\ \epsilon & \textit{otherwise} \end{cases}$$

The joinpoint model M_{AJ} consists of a list of joinpoint abstractions representing the callstack: $M_{\text{AJ}} = J^*_{\text{AJ}}$. The corresponding transfer function maintains the call stack and the current joinpoint:

$$T_{\text{AJ}}(\bar{j}, j) = \begin{cases} j \cdot \bar{j} & \text{if } j = C.m(\bar{C}) \\ \bar{k} & \text{if } j = \texttt{return} \text{ and } \bar{j} = k \cdot \bar{k} \end{cases}$$

[3] The actual values may be needed for the execution of an advice in AspectJ, but this work focuses on pointcuts only.

An Alpha-like Joinpoint Model. In Alpha, not only the types but also the dynamic values of the objects involved in a state are available when formulating pointcuts. Hence the joinpoints in Alpha are more fine-grained[4]:

$$J_{\text{Alpha}} = \{v.m(\bar{v}), \texttt{return}(v)\}$$

Furthermore, not only the callstack but the complete execution history is available for pointcuts. Accordingly, the abstraction mapping α_{AJ} is the projection on the current redex:

$$\alpha_{\text{Alpha}}(\sigma) = \begin{cases} v.m(\bar{v}) & \text{if } \sigma = (p, E[v.m(\bar{v})]) \\ \texttt{return}(v) & \text{if } \sigma = (p, E[\texttt{return}(v)]) \\ \epsilon & \textit{otherwise} \end{cases} .$$

The model values in this model are meant to keep the complete (abstracted) trace of the execution history. We will model this again as a list of joinpoints: $M_{\text{Alpha}} = J^*_{\text{Alpha}}$, together with a transfer function which accumulates all relevant joinpoint abstractions in the list: $T_{\text{Alpha}}(\bar{j}, j') = j \cdot \bar{j}$.

In contrast to the AspectJ-like model, `return` joinpoint abstractions are stored and not discarded. The reason is that Alpha pointcuts frequently reconstruct former callstacks from the history, hence this information is needed to evaluate such pointcuts.

2.4 Classification of Joinpoint Models

In the previous section, we saw two different joinpoint models for a small base language. In order to illustrate the design space of the combination of joinpoint abstraction and model definition, we will now look at some of typical cases of joinpoint models.

The space dimension The first axis of the design space of joinpoint models is concerned with how much information about the past can be stored in a model value. The simplest such joinpoint model is the *constant model*, where $T_{\text{const}}(m, j) = m$. In this model, the initial model value is present at every point in the program. This kind of model can be used to represent configuration values in a system or to pass the AST of the program or parts of it around. A constant model alone is not very useful, since the model value never changes, but it is useful if combined with other models, see later discussion about model constructions.

In contrast, *global models* are not restricted in the way they collect information about the execution. Given a fixed JPA, the model values of a global model are lists representing a view on the past execution and the transfer function is of the form $T_{\text{global}}(m, j) = f(j) \cdot m$ for some function f. The idea behind this definition is that the joinpoint model accumulates information (possibly transformed by f) from every state in the execution in a list. The most general global

[4] Again we omit field reads to illustrate how reductions can be ignored

model is the *full trace model*, which has $f = id_J$. The Alpha joinpoint model falls in this category.

A slightly more restricted model is a *bounded model*, where the size of the model (according to some size metric) is restricted by a function on the current state. The AspectJ joinpoint model falls in this category: The length of the stored list is bounded by the depth of the current callstack.

Finally, a *local model* is a joinpoint model that is restricted in the amount of information that it can keep in one model value. The canonical example is a model with $T_{\text{local}}(m, j) = j$, where the model only has access to the most recent state. The joinpoint value can of course be transformed by any function instead of the identity in T_{local}. The AspectJ pointcut language without the `cflow`-style pointcuts could be evaluated in terms of a local joinpoint model. The same holds for the breakpoint conditions typically found in debuggers.

Static and dynamic The other axis of the design space of joinpoint models is the kind of information that can be accessed. Here we distinguish joinpoint models whose only information about the current state is the lexical position of the current redex from joinpoint models that also refer to other dynamic information. Henceforth, a *static model* is one whose corresponding joinpoint abstraction function α has the form $\alpha(\sigma) = \text{loc}(\sigma)$, where loc is a function that retrieves the lexical position of the current redex. Such pointcut languages are particularly easy to implement, since every pointcut can be directly mapped to a set of locations in the code. An example is the pointcut language by Eichberg et al. [4]. In contrast, *dynamic* models are not restricted in the way they information about the state.

Product Models In most pointcut languages the joinpoints models are combinations of the classes of models described above. For example, even if a very generic trace based model may be available, there may be static source code information that is not part of the trace, like the subtype relation. This means that models will usually be cartesian products of different kinds of models whose components can be classified using the categories above.

The *product model* of two joinpoint models $\mathbf{M} = (M, T_M)$ and $\mathbf{M'} = (M', T_{M'})$ is $\mathbf{M_P} = \mathbf{M} \times \mathbf{M'}$, where the product of the transfer functions is defined componentwise as $T_{M_P}((m, m'), j) = (T_M(m, j), T_{M'}(m', j))$. Of course, the product construction can easily be generalized by replacing the pair constructor with an arbitrary function. Such a construction can be used to model dependencies between the joinpoint models in such a way that one model value influences the other one.

The categorization of joinpoint models (or their components) is not only useful when comparing different joinpoint models and pointcut languages but also in design and implementation of aspect oriented systems: the kind of mapping indicates which information must be stored or calculated in the runtime system when augmenting or implementing a system for a pointcut language.

3 Pointcuts

So far we have defined what we mean by joinpoint models and how these models are connected to the underlying operational semantics by abstraction of states into joinpoints. Based on these definition we will now describe what pointcuts are in our framework and how pointcut languages can be modeled.

3.1 Pointcuts and Matching

We have already stated that pointcuts are a means to identify points in the execution of a program that share a certain property. In our framework, the states of the program are not directly available but through the abstraction of joinpoint models. Thus pointcuts are modeled by stating on which joinpoint model values they match.

Definition 3 (Pointcut). *Let* $\mathbf{M} = (M, T_M)$ *be a joinpoint model. A pointcut* P *is a set of model values, i. e.,* $P \subseteq M$. [5] *Pointcut* P *is said to* match a model value m*, if* $m \in P$.

The definition of pointcut matching can be extended to execution traces, i. e., lists of the form $t = \sigma_1, \ldots, \sigma_n$ of states with $\sigma_i \longrightarrow \sigma_{i+1}$. To this end we have to define how model values are transformed over traces of states. We begin by lifting the transfer function to a function $T_M^* : M \times (J \cup \{\epsilon\})^* \to M \times (J \cup \{\epsilon\})$ that maps the transfer function over *abstract traces*, that is, over lists of joinpoint abstractions by $T_M^*(m, []) = (m, \epsilon)$ and

$$T_M^*(m, j \cdot \bar{j}) = \begin{cases} (T_M(m, j), j) & \text{if } j \neq \epsilon \text{ and length}(\bar{j}) = 0 \\ T_M^*(m, \bar{j}) & \text{if } j = \epsilon \\ T_M^*(T_M(m, j), \bar{j}) & \text{if } j \neq \epsilon \text{ and length}(\bar{j}) > 0 \end{cases}$$

This is a glorified fold of the transfer function over the abstract trace, which is a little more complex than a normal fold due to the fact that abstract traces may contain ϵ values, which are no valid arguments for the model transfer function T_M. However, we cannot simply ignore them; if a trace finishes on ϵ, then the model value assigned to this trace is the result of an earlier state in the execution, and a pointcut should match only at that earlier state and not on each subsequent state, that is mapped to ϵ. Thus we keep track of the last joinpoint that has been used to calculate the model value and use ϵ to indicate, that no pointcut should match this trace.

As T_M^* operates on lists of joinpoint abstractions, we need to lift traces, which consist of states, to abstract traces, which consist of the corresponding joinpoint abstractions. We define this operation $\alpha_J^*(\bar{\sigma}) = \text{map}(\alpha_J, \bar{\sigma})$. Based on this we define what it means that a pointcut matches a trace.

[5] Pointcuts can be identified with *predicate functions* by looking at the characteristic function χ_P.

Definition 4 (Pointcut Matching on Traces). *Let M be a joinpoint model with joinpoint abstraction J. A* pointcut P matches a trace t *with initial model value m_0, if $T_M^*(m_0, \alpha_J^*(t)) = (m, j)$ with $j \in J$ and $m \in P$.*

3.2 Pointcut Languages

The means by which a pointcut is described is defined by a pointcut language. A pointcut languages consists of syntax – a set of valid expressions – and a semantics, mapping syntactical constructs to their meaning in terms of pointcuts.

Definition 5 (Pointcut Language). *Let $\mathbf{M} = (M, T_M)$ be a joinpoint model. A pointcut language $\mathbf{L}$ over $\mathbf{M}$ is a set L of language expressions called* pointcut designators *together with a semantics function $[\![\cdot]\!]_L : L \to \mathcal{P}(M)$, which maps pointcut designators to their* extension*, i.e., the set of model values on which the corresponding pointcut matches.*
We define the extension of a pointcut language *as $[\![\mathbf{L}]\!] = \{[\![\pi]\!] \mid \pi \in L\}$.*

In any non-trivial pointcut language, there will be multiple pointcut designator with the same extension – in most cases even infinitely many. For example, in AspectJ one can find arbitrary many pointcut designators matching all possible method calls: `call(*.*(..))`, `call(*.*(..)) && call(*.*(..))`, and so forth. It is further noteworthy that some of the extension sets will be infinitely large, because we do not fix a program in the definition of pointcut languages. For example, if an identifier – like a method name – is part of the model values and a pointcut designator does not impose any restrictions on this identifier, the extension of this pointcut designator will comprise model values for each identifier expressible in the language at hand.

It is, however, an interesting property of a pointcut language, which extensions are finite for *any* or *some* fixed program. For example, in the case of identifiers, extensions will be finite (at least with respect to the identifier) because the set of identifiers is finite for any given program. On the other hand, in pointcut languages over joinpoint models containing the whole execution history, many pointcut designators will have infinite extensions, because there are infinitely many possible traces satisfying the constraints given in the pointcut designator.

3.3 Example Pointcut Languages

We have already given the definition of the pointcut model of Alpha and AspectJ like languages defined over a small OO base language. We will discuss how to compare pointcut languages next and for that purpose we want to discuss briefly the pointcut languages of these two approaches. Due to place constraints we will not give a complete definition of the syntax and semantics in terms of our framework.

AspectJ pointcuts are basically built from a set of primitive pointcut expressions using boolean the operators `&&` (and), `||` (or) and the higher-order pointcut `cflow`, which is true if its argument denotes a joinpoint on the callstack. For our purpose, only the primitive pointcut `call(R C.m(a))` is important. In this pointcut `R` is the return type, `C` the class name, `m` the method name and `a` argument. For each of these parameters, wildcards can be used.

Alpha pointcuts are basically Prolog queries over a representation of the execution trace. This representation contains facts for each joinpoint that occurred in the execution, paired with a time stamp. These time stamps can be used to relate the joinpoints and to define abstractions like `cflow`. Since the programmer can also provide his/her own predicates, Alpha pointcuts can compute arbitrary functions over the trace representation.

Both of these approaches make the quantification over the joinpoint model implicit. In order to come up with a semantics in the form of Def. 5, we would have to make this relation explicit by presenting the semantics as predicate functions over the joinpoint model and then use these predicate functions as characteristic functions for the pointcut extension.

4 Comparing Pointcut Languages

In this section we will develop a methodology to systematically compare pointcut languages. Since we introduced three dependent layers – joinpoint abstractions, joinpoint models and pointcut languages – we have to define comparison on the lower layers first, to make constructs on the higher layers comparable. Thus we start by describing, what it means that one of two joinpoint abstractions is more abstract. Then we talk about the relation between joinpoint models over the *same* joinpoint abstraction. Finally we show how pointcut languages can be compared and how constructive proofs can be constructed for one language being more/less expressive than an other.

4.1 Comparison of Joinpoint Abstractions

We begin by describing how joinpoint abstractions can be compared. We assume the same base language semantics for both abstractions, because we want to talk about the way that the abstractions act on the *same* states. We define a joinpoint abstraction to be *more abstract* than another (more detailed) joinpoint abstraction, if it identifies joinpoints of the more detailed model.

Definition 6 (Abstractness of Join Point Abstraction). *Let* $\mathbf{J_1} = (J_1, \alpha_{J_1})$ *and* $\mathbf{J_2} = (J_2, \alpha_{J_2})$ *be joinpoint abstractions over* Σ*. Then* $\mathbf{J_2}$ *is* more abstract *than* $\mathbf{J_1}$ *if there exists a surjective mapping* $\delta : J_1 \rightarrow J_2$*, such that* $\delta \circ \alpha_{J_1} = \alpha_{J_2}$*.*

Consider the joinpoint abstractions J_{AJ} and J_{Alpha} from Sec. 2.3. According to Def. 6, the joinpoint abstraction J_{AJ} is obviously more abstract than J_{Alpha}, because its elements contain only type names, while the elements of J_{Alpha} contain values. We show this formally by constructing $\delta : J_{\text{Alpha}} \rightarrow J_{\text{AJ}}$ as follows:

$$\delta(j) = \begin{cases} \text{type}(v).m(\text{type}(\bar{C})) & \text{if } j = v.m(\bar{v}) \text{ and } C_i = \text{type}(v_i) \\ \texttt{return} & \text{if } j = \texttt{return}\ (v) \\ \epsilon & \text{otherwise} \end{cases}$$

and showing that δ satisfies the condition given in Def. 6:

$$\begin{aligned} \delta \circ \alpha_{Alpha}(E[v.m(\bar{v})]) &= \delta(v.m(\bar{v})) = \text{type}(v).m(\text{type}(\bar{v})) \\ &= \alpha_{AJ}(E[v.m(\bar{v})]) \\ \delta \circ \alpha_{Alpha}(E[\texttt{return}(v)]) &= \delta(\texttt{return}(v)) = \texttt{return} \\ &= \alpha_{AJ}(E[\texttt{return}(v)]). \end{aligned}$$

All other expressions are mapped to ϵ by both joinpoint abstraction mappings, so these are the only two cases to consider. Thus we have shown that $\delta \circ \alpha_{\text{Alpha}} = \alpha_{\text{AJ}}$, and therefore that the call stack based model is more abstract than the trace model.

4.2 Comparison of Joinpoint Models

When comparing joinpoint models, one important problem is that the underlying joinpoint abstractions may be different. In order to find a common joinpoint abstraction to compare both models, we can use the joinpoint abstraction translation δ to compare both models on the more abstract joinpoints. Given this common joinpoint abstraction, the key idea is that values of the more expressive (richer) joinpoint model contain enough data to reconstruct values of the less expressive model. This means that there exists a mapping between the model values which is "compatible" with the model transfer function.

Definition 7 (Sub-Model). *A joinpoint model* $\mathbf{M_1} = (M_1, T_1)$ *is a sub-model of* $\mathbf{M_2} = (M_2, T_2)$*, if both have the same joinpoint abstraction* J *and there exists a surjective function* $\beta : M_2 \to M_1$*, such that*

$$\forall j \in J, m \in M_2.\ \beta(T_2(j, m)) = T_1(j, \beta(m)).$$

The mapping β identifies all elements of $\mathbf{M_2}$ that can not be distinguished in $\mathbf{M_1}$ by mapping them to the same element. The condition on β requires the mapping to be consistent with the model transfer function. If β is a bijection then both models are equal (up to renaming), otherwise $\mathbf{M_1}$ is a proper sub-model of $\mathbf{M_2}$.

For example, consider again the call stack and trace based models from Sec. 2.3. We define $\beta : M_{\text{Alpha}} \to M_{\text{AJ}}$ as follows:

$$\beta([]) = [], \quad \beta(j \cdot \bar{j}) = \begin{cases} \delta(j) \cdot \beta(\bar{j}) & \text{if } j = v.m(\bar{v}) \\ \bar{k} & \text{if } j = \texttt{return}(v) \text{ and } \beta(\bar{j}) = k \cdot \bar{k} \end{cases}$$

It is easy to prove that the result of β is indeed the corresponding call stack. That β is correct with respect to the definition can be seen from the fact, that δ is correct and by simulating the semantics for one step.

4.3 Criteria for the Comparison of Pointcut Languages

In the following we compare pointcut languages, and similar to the joinpoint model comparisons, we assume that both languages are defined over the same joinpoint model, since we can translate the richer joinpoint model into the less expressive and compare the translated languages over the less expressive model.

Since pointcut languages consist of both a syntax and a semantics we can compare pointcut languages with respect to both. First, we ignore syntax and concentrate on the extensions of the languages, that is, the sets of model values that are extensions of pointcuts in the languages. Later we describe how languages can be compared with respect to syntax and finally we give a third formulation based on the the ability of a language to distinguish program executions.

Using the definition of the extension of a pointcut language (Def. 5), we can formalize our first comparison criterion. The idea is that a language $\mathbf{L_1}$ is a sub-language of $\mathbf{L_2}$, if the extension of $\mathbf{L_2}$ contains every pointcut that is in the extension of $\mathbf{L_1}$:

Definition 8 (Expressiveness of Pointcut Languages). *Let* $\mathbf{L_1}$ *and* $\mathbf{L_2}$ *be two pointcut languages over the same model* M*. Then,* $\mathbf{L_1}$ *is less expressive, if* $[\![\mathbf{L_1}]\!] \subseteq [\![\mathbf{L_2}]\!]$ *and* $\mathbf{L_1}$ *is* proper *less expressive than* $\mathbf{L_2}$*, if* $[\![\mathbf{L_1}]\!] \subset [\![\mathbf{L_2}]\!]$*.*

This definition means that if there is a pointcut designator $\pi_1 \in L_1$ (i. e. $[\![\pi_1]\!] \in [\![\mathbf{L_1}]\!]$) then there exists a pointcut designator $\pi_2 \in L_2$ such that $[\![\pi_1]\!] = [\![\pi_2]\!]$. Consider, for example, the Alpha pointcut `calls(R, 'm', V), typeof(R, 'C'), odd(V)` [6], which matches all traces ending on a call of a method m on instances of class C, where the argument is an odd integer. To compare the extension of this pointcut to AspectJ pointcuts, we need to embed the AspectJ pointcut language into the Alpha pointcut model. To this end we identify a pointcut $P \subseteq M_{\mathrm{AJ}}$ with the Alpha pointcut $\beta^{-1}(P) = \{m \in M_{\mathrm{Alpha}} \mid \beta(m) \in P\}$. Since we cannot reason about values in AspectJ, $\beta^{-1}(P)$ contains traces ending in calls with *every possible argument value.* Hence, there is no AspectJ pointcut that corresponds to our Alpha pointcut and thus AspectJ is not equally expressive as Alpha. However, since it is possible to reason about types of values in Alpha, every AspectJ pointcut can be expressed in Alpha. This means that AspectJ is a proper less expressive language than Alpha.

The expressiveness of a pointcut language can also be expressed on the level of syntax. On this level, a less expressive or *sub-language* is characterized by a mapping between the pointcut designators of the two languages. This mapping maps each pointcut designator of the less expressive language to one of the more expressive language:

Definition 9 (Sub-Language). *Let* $\mathbf{L_1}$ *and* $\mathbf{L_2}$ *be two pointcut languages over the same model* M*.* $\mathbf{L_1}$ *is a sub-language of* $\mathbf{L_2}$*, if there exists a mapping of pointcut designators* $\iota : L_1 \hookrightarrow L_2$ *with:* $\forall \pi \in L_1.\ [\![\pi]\!] = [\![\iota(\pi)]\!]$*.*

[6] We assume that some representation of integers is available and write them as numbers to keep the examples short.

Of course, the extension of the pointcuts must be invariant under the syntactic mapping, which is what the condition imposed on ι ensures. In a sense, this comparison criterion is more useful, because it requires to construct a concrete syntactical mapping. In practice, this mapping can be used for the emulation of the sub-language in an implementation of the super-language.

We have already mentioned that there is an Alpha pointcut for every possible AspectJ pointcut, which means that we can construct a mapping $\iota : \mathbf{L}_{\text{AspectJ}} \rightarrow \mathbf{L}_{\text{Alpha}}$ by translating the primitive and `cflow` AspectJ pointcuts into the corresponding Prolog terms and translate the boolean operators into logical connectives in Prolog.

On the other hand it is often desirable to have a constructive proof that a language is *not* a sub-language of another language. With the above criteria, this can only be established by proving the non-existence of such a mapping. The definition of pointcut matching over traces (Def. 4) allows for another characterization of pointcut languages. A pointcut can be identified with the set of traces on which it matches for a given initial model value. Thus we can classify pointcut languages with the traces that are separable by any pointcut designator belonging to the language.

First note that this set of traces is restricted by the joinpoint abstraction: each abstract trace can be the abstraction of several traces. The set of traces that correspond to an abstract trace t_α is: $(\alpha_J^*)^{-1}(t_\alpha) = \{t \mid \alpha_J^*(t) = t_\alpha\}$. Thus pointcuts can only be compared by the abstract traces that they are able to distinguish. We will call two traces t_1 and t_2 *separable by a pointcut* π, if there is a model value m on which $T_M^*(\alpha_J^*(t_1), m)$ matches π iff $T_M^*(\alpha_J^*(t_2), m)$ does not match π. This notion of separability can be used to define an equivalence relation on traces that identifies all traces which can be separated by a given pointcut language:

Definition 10 (Separability Equivalence). *Let* $\mathbf{L}$ *be a pointcut languages over a model* $\mathbf{M} = (M, T_M)$*. The separability equivalence relation* $\equiv_L$ *for the language* $\mathbf{L}$ *is defined as follows* $t_1 \equiv_L t_2 :\Leftrightarrow$

$$\forall \pi \in L, m \in M.\ T_M^*(m, \alpha_J^*(t_1)) \in ([\![\pi]\!] \times J) \Leftrightarrow T_M^*(m, \alpha_J^*(t_2)) \in ([\![\pi]\!] \times J).$$

Thus we can give an alternative definition for pointcut language expressiveness based on the pointcut languages' ability to separate sets of model values:

Definition 11 (Precision of Pointcut Language). *Let* $\mathbf{L_1}$ *and* $\mathbf{L_2}$ *be two pointcut languages over the same model* M*.* $\mathbf{L_1}$ *is less precise than* $\mathbf{L_2}$*, if* $\equiv_{L_2} \subseteq \equiv_{L_1}$*.* L_1 *is* proper *less precise than* L_2*, if* $\equiv_{L_2}$ *is a proper subset of* $\equiv_{L_1}$*.*

From the previous examples we see that Alpha is much more precise than AspectJ, because it is possible to refer to runtime values in the pointcut. This is, however, not due to the lack of values in our modeling of AspectJ: Even if we had values instead of types, AspectJ cannot distinguish traces that differ in arbitrary computable properties, like the sum of all arguments, the structure of the heap (reachability) and so forth.

4.4 A Methodology for Language Comparison

Using the criteria above we propose the following methodology for comparing pointcut languages: First, we have to ensure, that the two pointcut languages we want to compare are defined over the same joinpoint model and thus over the same joinpoint abstraction. The latter can be achieved by using δ to translate detailed joinpoints into abstracted ones. Then we construct the β mapping from the more expressive model to the less expressive and use it perform the comparison either on the less expressive model or on the more expressive model using β^{-1} as in the examples.

To show that one language is a sub-language of/less-expressive than another language, we construct a mapping from pointcut designators of the sub-language to pointcut designators of the richer language. To prove that a language $\mathbf{L_1}$ is not a sub-language of another language $\mathbf{L_2}$, two traces have to be provided that the language $\mathbf{L_1}$ can separate, but $\mathbf{L_2}$ can not.

In our example comparison we have seen that there are two different reasons, why AspectJ is less expressive than Alpha: The lack of values in the joinpoint model and the simpler computational expressiveness. Care must be taken not to confuse these different sources of expressiveness by comparing the languages with respect to both the more expressive and the less expressive model.

5 Discussion

Finally we will give an outlook on how our formalization of pointcuts and models is connected to typical issues found in the implementation of programming languages and runtime monitoring systems.

5.1 Shadows and Optimization

We have identified pointcuts with sets of joinpoint model values. By the joinpoint abstraction, each of these model values can be identified with one or more traces in the program execution. To be precise, we can identify each model value $m \in M$ with the set T_m of pairs of traces and initial values that produce m given by: $T_m = \{(t, m_0) \mid (\exists j \in J)\ T^*_M(m_0, \alpha^*_J(t)) = (m, j)\}$. For a pointcut π, the set $S_\pi = \bigcup_{m \in [\![\pi]\!]} \{t \mid (\exists m_0)\ (t, m_0) \in T_m\}$ is the set of traces that may lead to a model state that the pointcut matches.

This set is of particular interest, if there is a function *loc* mapping states to elements of the program representation. For example, when using labeled expressions, the labels can be used to identify the source code (or, program) element that is currently evaluated. These source code locations are typically called *shadows* of the pointcut and the set of residues

$$\mathrm{Res}_\pi(l) = \{(\sigma_0, \ldots, \sigma_n) \in\ S_\pi \mid loc(\sigma_n) = l\},$$

can be thought of as defining the dynamic test to be performed at the labeled element to decide if the pointcut matches.

Shadows and the corresponding dynamic checks are important concepts in the implementation and optimization of pointcut languages. However, without further information about the structure of states (and, therefore, of traces), the elements of $\mathrm{Res}_\pi(l)$ are not a very good descriptions of the dynamic checks, because they simply enumerate all matching traces. But if it is possible to exploit the semantics of the base language and knowledge about the joinpoint model to compute a set T_l of approximated traces for the location l, we can eliminate all traces from $\mathrm{Res}_\pi(l)$ which are no approximations, possibly yielding a much smaller set of traces. In the extreme case the remaining set could be empty. In this case we know that the pointcut can never match an execution trace reaching this location and we do not have to check for matching at runtime. But even in the case of a non-empty result, an optimization is possible if we are able to find a "more efficient" pointcut expression π' with $\mathrm{Res}_\pi(l) \cap T_l = \mathrm{Res}_{\pi'}(l) \cap T_l$.

These optimizations are too dependent on the exact details of the base language, joinpoint model and pointcut language to be described in this work, but a specialization of our framework for a particular language is likely to yield concrete and useful optimization strategies.

5.2 Observational Equivalence

Two expressions are *observationally equal*, if they cannot be distinguished by any context in which they are inserted. Observational equivalence is obviously important both in terms of program understanding and optimization.

If we consider a runtime monitoring system over a fixed language, the observational equivalence relation will in general become smaller, because a pointcut observing the program execution can suddenly distinguish expressions that would otherwise be observationally equivalent. For example, a call to a method that returns a constant cannot be replaced with this constant, because a trace model could distinguish these terms.

Let us assume that the expressions M and N are observationally equivalent in the unobserved program. We can now look at the modified observational equality relation on different levels. If we allow arbitrary models, we have to consider all traces that the evaluation of the terms may produce. If the traces in all contexts are equal for both expressions, then the expressions are also observationally equal with respect to the runtime monitoring system. If we fix the model, we have more observationally equal expressions, but we still have to check all possible traces in all contexts. Finally, with a fixed pointcut language, we have to check if the terms have *separable* traces in at least one context, otherwise they are observationally equivalent. This means that we can find better approximations for observational equivalence, if we have a compact description of separability for the given pointcut language.

A typical application of identifying observationally equivalent expressions is in program optimization. Like with shadows, in this generality we cannot give concrete optimizations. However, for a fixed base and pointcut language, the separability relation may have a simple approximation that allows to derive simple static analyses to approximate observational equivalence.

5.3 Advice and Context Binding

Using pointcuts to trigger additional functionality is only one possible application of pointcuts, but it is useful in many contexts, like debugging and tracing, enforcements of trace and security properties, and aspect-oriented modularization. In this section we show how a "aspect-aware" semantics can be defined on top of a base semantics and a pointcut language over this semantics. To make this elaboration a bit less abstract, we use the convention from our former examples and and define the states of the base semantics as pairs $(p, E[e])$ of the program representation and a context E with a redex e.

We give a semantics which augments the original semantics by adding the required context, a syntactical element for proceeding with the evaluation of the original state of execution, and derivation rules for pointcut matching and advice evaluation to the semantics of the underlying language. These extensions describe when and how pointcuts are evaluated and how they trigger the evaluation of advice functionality.

The additional functionality is specified in *aspects*, which bind a piece of code to the pointcut which describes when to execute that code. There are different ways to introduce the additional functionality: *before*, *after* or *instead* (often called *around*) the original expression. In our model we use the *around* strategy, because both other strategies can be expressed in terms of *around*.

It is not sufficient, however, to simply use a pair of a pointcut and an expression to model aspects, because aspects may bind information from the model value and they may choose to proceed with a modified expression instead of the original one. Taking this possibilities into account we model aspects as the set

$$\mathcal{A} = \mathbf{L} \times \underbrace{(M \to \mathcal{E})}_{\text{bind}} \times \underbrace{(M \to \mathcal{E} \to \mathcal{E})}_{\text{target}} .$$

The *bind* function describes how the context is used to define the additional functionality. For example, the bind function $C.m(\bar{C}) \mapsto \texttt{print}(m); \texttt{proceed}$ prints the method name of the intercepted method before proceeding with the call. The *target* is used to modify how to proceed with the evaluation. It takes the model value *at the point `proceed` is about to be evaluated* and the original expression and produces the expression that has to be evaluated instead of `proceed`.

To model this augmented semantics, we fix a pointcut language $\mathbf{P}$ with execution model $\mathbf{M}$ and joinpoint abstraction $\mathbf{J}$. The syntax of the base language $(\mathcal{E})$ is extended by the expression `proceed`, which can be used in advice to resume the evaluation of the expression that lead to the advice activation. Since the definition of the base language's syntax will in most cases be a recursive constructor, the definition of the extended syntax depends on the base syntax and can not be defined in a general way. We assume that such a syntax $\mathcal{E}'$ can be constructed from $\mathcal{E}$ and that this set contains all the base expressions and additionally – at least – the expression `proceed`.

We define the lifted semantics as follows: the new states are of the form

$$\frac{a = (\pi, b, t) \qquad m' = T_M(\alpha_J(\sigma), m) \qquad m' \in [\![\pi]\!]}{(m, a \cdot \bar{a}, \bar{e}, (p, E[e])) \longrightarrow' (m, a \cdot \bar{a}, e \cdot \bar{e}, (p, E[\mathrm{b}(m')]))} \quad (\textsc{Advice App})$$

$$\frac{(m, \bar{a}, \bar{e}, \sigma) \longrightarrow' \sigma' \qquad e \neq \texttt{proceed}}{(m, a \cdot \bar{a}, \bar{e}, \sigma) \longrightarrow' \sigma'} \quad (\textsc{Cong})$$

$$\frac{\forall (\pi, b, t) \in \bar{a}.\ T_M(\alpha_J(\sigma), m) \notin [\![\pi]\!]}{(m, \bar{a}, e \cdot \bar{e}, (p, E[\texttt{proceed}])) \longrightarrow' (m, \bar{a}, \bar{e}, (p, E[t(m, e)]))} \quad (\textsc{Proceed})$$

$$\frac{\sigma \longrightarrow \sigma' \qquad \sigma = (p, E[e]) \qquad e \neq \texttt{proceed} \qquad \forall (\pi, b, t) \in \bar{a}.\ T_M(\alpha_J(\sigma), m) \notin [\![\pi]\!]}{(m, \bar{a}, \bar{c}, \sigma) \longrightarrow (T_M(\sigma, m), \bar{a}, \bar{c}, \sigma')} \quad (\textsc{Base})$$

Fig. 1. Evaluation Rules for the Lifted Semantics

$$\Sigma' = M \times [\mathcal{A}] \times [e] \times \Sigma.$$

An element $(m, \bar{a}, \bar{e}, \sigma) \in \Sigma'$ consists of the current model value m, a list $\bar{a}$ of aspects, a list $\bar{e}$ of expressions whose evaluation has been "shadowed" by invocation of an aspect, and the original state σ.

The evaluation relation $\longrightarrow' \subset \Sigma' \times \Sigma'$ is defined by the rules in Figure 1. The (Advice App) rule explains how matching advice is applied. Rule (Cong) allows to use other aspects than the head of the list. This rules introduces a non-determism for advice application. Note that we did not model multiple aspects to match at the same state. The third rule describes the semantics of the new expression `proceed`, and the final rule embeds the base semantics.

6 Related Work

Many formal definitions of the semantics of aspect oriented languages have been proposed to clarify the semantics concepts like advice application, aspect precedence, and proceed. For example, Masuhara et al. [13] present a model that explains compilation and optimization of an AspectJ-like language. The model is based on an interpreter which is partially evaluated to explain issues like shadow finding and removal of runtime checks which are unnecessary. The resulting model is not as general as our formalization, because it is tailored to AspectJ. This model (and similar ones for other pointcut languages) can be used to describe the semantic of AspectJ-pointcuts in our formal model, because it defines the characteristic functions for pointcut extension. This and similar approaches use fixed base and pointcut languages; a general exploration and organization of the pointcut language design space is not in the scope of these works.

Another branch of related work is the development of meta-models and ontologies for pointcut-advice (PA) languages [2, 3]. These models are used to identify the parts in PA-languages in a very general way that can be used as a basis for implementations of aspect-oriented systems. Although these models contain informal representations of pointcuts, pointcut context (similar to our joinpoint models), as well as a distinction between static and dynamic context, these entities do not have a formally specified meaning and are hence not amenable to constructive comparisons between pointcut languages.

Störzer and Hanenberg [16] give a categorizations of pointcut language constructs with respect to a fixed set of three different model classes: specification based pointcut languages, which are similar to our constant models, state-based constructs, which are similar to our local models, and so-called progress-based pointcut constructs, which are a subset of our global (trace-based) models. In contrast to our work, there is no notion of joinpoint abstraction or pointcut language expressiveness, and no methodology to compare pointcut languages.

Masuhara and Kiczales [12] present a comparison framework in which they evaluate four different AOP languages. The focus of that work is to characterizing and comparing the *weaving process* that these languages use. This is in contrast to our framework, which focuses only on the language part and ignores advice and weaving, thus making the framework applicable in "non-AOP" systems.

7 Conclusions

We have presented a formal framework for the classification and comparison of pointcut languages. This model gives precise meaning to joinpoints, pointcuts and pointcut languages. The most important usage of this framework is a methodology for the comparison of pointcut languages, which is based on giving constructive proofs for sub-language relations. These proofs can be used to construct embeddings of pointcut languages into more expressive ones, or to characterize the reason why one language is less expressive than another. Furthermore, our framework illustrates the design dimensions and corner cases of pointcut language design and implementation, and hence we hope that it can be used as guidance in both the design of new pointcut languages and the development of efficient compilation techniques for pointcut languages.

We consider our framework both as a starting point to improve the modeling, design and implementation of pointcut languages and as way to combine the existing approaches to give semantics to particular pointcut languages.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA '05: Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 345–364, New York, NY, USA, 2005. ACM Press.

2. C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *VMIL '07: Proceedings of the workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, New York, NY, USA, 2007. ACM Press.
3. C. Bockisch, M. Mezini, K. Gybels, and J. Fabry. Initial definition of the aspect language reference model and prototype implementation adhering to the language implementation toolkit architecture. Technical report, AOSD Europe Deliverable, Technische Universität Darmstadt, 2007.
4. M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In W.-N. Chin, editor, *APLAS '04: Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*, Lecture Notes in Computer Science, pages 366–382, Taipei, Taiwan, November 2004. Springer-Verlag.
5. E. Hajiyev, M. Verbaere, and O. de Moor. codeQuest: Scalable source code queries with datalog. In D. Thomas, editor, *ECOOP '06: Proceedings of the European Conference on Object-Oriented Programming*, pages 2–27. Springer-Verlag, 2006.
6. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In A. M. Berman, editor, *OOPSLA '99: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 132–146, N. Y., 1999.
7. D. Janzen and K. De Volder. Navigating and querying code without getting lost. In M. Aksit, editor, *AOSD '03: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 178–187, New York, NY, USA, 2003. ACM Press.
8. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP '01: Proceedings of the European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353, 2001.
9. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97: Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.
10. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In R. P. G. Ralph E. Johnson, editor, *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, volume 40, pages 365–383, New York, NY, USA, 2005. ACM.
11. H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. In N. Kobayashi, editor, *APLAS '06: Proceedings of the 4th Asian Symposium on Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2006.
12. H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *ECOOP'03: Proceedings of the European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 219–233. Springer-Verlag, 2003.
13. H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *CC '03: Proceedings of 12th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 46–60. Springer-Verlag, 2003.

14. K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In A. P. Black, editor, *ECOOP'05: Proceedings of the European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer-Verlag, 2005.
15. G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In D. F. Bacon, editor, *OOPLSA'07: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 535–552. ACM, 2007.
16. M. Stoerzer and S. Hanenberg. A classification of pointcut language constructs. In *SPLAT'05: Proceedings of the workshop on Software-Engineering Properties of Languages and Aspect Technologies*, 2005.
17. R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In R. N. Taylor and M. B. Dwyer, editors, *FSE '04: Proceedings of the International Symposium on Foundations of Software Engineering*, pages 159–169, New York, NY, USA, 2004. ACM.
18. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.