# Expressive Pointcuts for Increased Modularity

Klaus Ostermann and Mira Mezini and Christoph Bockisch

Darmstadt University of Technology, D-64283 Darmstadt, Germany
{ostermann,mezini,bockisch}@informatik.tu-darmstadt.de

**Abstract.** In aspect-oriented programming, pointcuts are used to describe crosscutting structure. Pointcuts that abstract over irrelevant implementation details are clearly desired to better support maintainability and modular reasoning.
We present an analysis which shows that current pointcut languages support localization of crosscutting concerns but are problematic with respect to information hiding. To cope with the problem, we present a pointcut language that exploits information from different models of program semantics, such as the execution trace, the syntax tree, the heap, static type system, etc., and supports abstraction mechanisms analogous to functional abstraction. We show how this raises the abstraction level and modularity of pointcuts and present first steps toward an efficient implementation by means of a static analysis technique.

## 1 Introduction

In aspect-oriented programming (AOP for short), pointcuts are predicates that identify sets of related points in the execution of a program, where to execute behavior pertaining to crosscutting concerns. Given an aspect that modularizes a crosscutting concern, its pointcuts serve as the interface between the crosscutting concern and the rest of the system. As such, the abstraction level at which these predicates are expressed directly affects the robustness of the design in the presence of change. Separation and localization of concerns into individual units is a major feature of modular design - providing interfaces that absorb local changes is another, equally important, feature.

It has been indicated elsewhere that a pointcut that merely enumerates relevant points in the execution by their syntactic appearance in the program code is fragile w.r.t. changes in the code [15, 20]. In this paper, we investigate the issue in more depth: We compare object-oriented (OO for short) and aspect-oriented (AO for short) designs of an exemplary problem with respect to their capability to remain stable in the presence of change. We observe that with current pointcut languages one can indeed separate crosscutting concerns into their own modular units, but the resulting design does not actually perform much better in terms of absorbing changes than the OO design which does not modularize the crosscutting concerns. This reduces the power of aspects to merely supporting pluggability of crosscutting concerns, leaving out of the reach another important modularity principle: Information hiding [30].

To cope with the problem, this paper proposes a pointcut language that allows to specify pointcuts at a high-level of abstraction by providing (a) different *rich models of the program semantics* and (b) *abstraction mechanisms* analogous to functional abstraction. The key insight is that various models of program semantics are needed to enable

reasoning about program execution. For example, the abstract syntax tree (AST) alone is not a very good basis for high-level pointcuts because it is a very indirect representation of the program execution semantics that makes it intractable to specify dynamic properties.

We propose to base the pointcut language on a *combination* of models of the programm's semantics. In this paper, we concentrate on four such models: The AST, the execution trace, the heap, and the static type assignment; if needed, other models such as a profiling or a memory consumption model could be added. Pointcuts in our approach are logic queries over the aforementioned models.

We have implemented a prototype of this approach as an interpreter for a small statically typed AO language, called ALPHA[1]. Pointcuts in ALPHA are logic queries written in Prolog [36]; they operate on-line over databases representing the aforementioned models of the program semantics. We show how AO designs expressed in this language can be made robust against various kinds of changes.

We also present a technique for an efficient implementation of our approach that is based on the notion of *join point shadows* [17]. The shadow of a dynamic join point is a code structure (expression, statement or block) that statically corresponds to an execution of the dynamic join point. The idea is to compute the shadows of pointcuts off-line by a static analysis of pointcuts and to evaluate or extend dynamic semantic models only at these statically computed shadows. Our analysis is different from previous approaches in this direction [17, 28, 35] in that it works on a much more powerful and open pointcut language.

Some concepts used in our approach have also been discussed elsewhere. For example, logic queries have been used in other approaches [15, 19, 37]. The unique contribution of our proposal compared to related work is twofold. First, we present a detailed study of the disadvantages of most current pointcut languages. Second, the openness of the pointcut language, the ability to combine different program models, and the incorporation of the execution trace and heap together with the abstraction mechanisms of a Prolog-like language is also unique. We will give a detailed account of the contribution of this paper and the relation to other works after the technical presentation.

The remainder of the paper is organized as follows. Sec. 2 motivates the need for better pointcut languages by a study of the robustness of aspect-oriented programs. Sec. 3 introduces the ALPHA programming language. Sec. 4 presents some examples in ALPHA and analyzes them in the light of the problems identified in Sec. 2. Sec. 5 describes the static analysis technique. Sec. 6 elaborates on the contribution of this paper in comparison to related work. Sec. 7 describes future work and concludes. The appendix contains different specifications of Prolog constructs that are used in various places but whose specification is not necessary to follow the paper.

## 2 Pointcuts and Modularity

In this section, we identify the limitations of current pointcut languages by means of an example problem. We focus on AspectJ's pointcut-advice mechanism [21] first; other

[1] Source code is available at [2].

pointcut languages will be discussed in the section on related work. We present an object-oriented (OO) and an aspect-oriented (AO) solution to the problem and compare them w.r.t robustness in the presence of change.

### 2.1 Example problem and its OO and AO solutions

The example problem is about modeling a hierarchy of graphical objects like points and lines which can be drawn on display objects; each display has a list of figures shown in it. The solution should ensure that the state of figure elements and their corresponding views on active displays is kept synchronized by having displays be updated when the state of figure elements changes.

An OO solution for the problem that applies the observer pattern [14] is schematically shown in Fig. 1[2]. To avoid unnecessary updates, the solution supports what we call *object precision* and *field precision*. By object precision we mean that an update to a figure element triggers a repaint only on those displays on which the figure element is visible; in general, there are multiple different display objects active, whereby every figure element is visible only on a (possibly empty) subset of all displays. For this purpose, each figure in Fig. 1 maintains an observer list with the displays it is shown in, if any; when a figure `f` is added to a display, the display is added to the list of `f`'s observers as well as to the observer lists of `f`'s children; e.g., showing a line on a display will cause the display to be an observer of the line as well as of its start and end points. If a figure `f1` is not anymore a child of another figure, `f2`, the observers that `f1` inherited from `f2` are removed from the list of `f1`'s observers[3].

By field precision we mean that only changes of the fields that contribute to the graphical representation of a figure element should trigger display updates. The set of the fields affecting the draw behavior generally depends on the dynamic control flow and cannot be determined statically. Hence, it is not always easy to ensure field precision especially if the system is complex. In the `Line` class in Fig. 1, it is easy to see that the field `name` is never involved in the drawing behavior and that `enable`, `start` and `end` are potentially read in the control flow of `draw`. Of the latter variables, only `enabled` is always read - hence, a change to it always triggers a notification of the observers; fields `start` and `end` are only read if `enabled` is true. Hence, changes to `start` or `end` trigger a notification only if `enabled` is true (see comments on the methods `notifyObserversUnconditional()`, `notifyObservers()` and `setEnabled(...)` in class `FigureElement`, and `Line.setEnd(...)` in Fig. 1).

A functionally equivalent AO solution of the problem is schematically shown in Fig. 2. This solution factors the observer management fields and methods out of the figure element classes into the aspect using the inter-type declaration mechanism of AspectJ[4]. The aspect defines three pointcuts. The pointcut `addFigure` captures any call

[2] Complete code for all examples and our ALPHA interpreter are available at [2].

[3] If figures are shared by several parent-figures, reference counters are associated with observers and an observer is actually removed from an observer list, only if its reference counter is zero.

[4] The observer implementation proposed by Hannemann and Kiczales [16] uses hashtables instead of introductions in order to increase the reusability of aspects, but this does not affect the discussion in this paper.

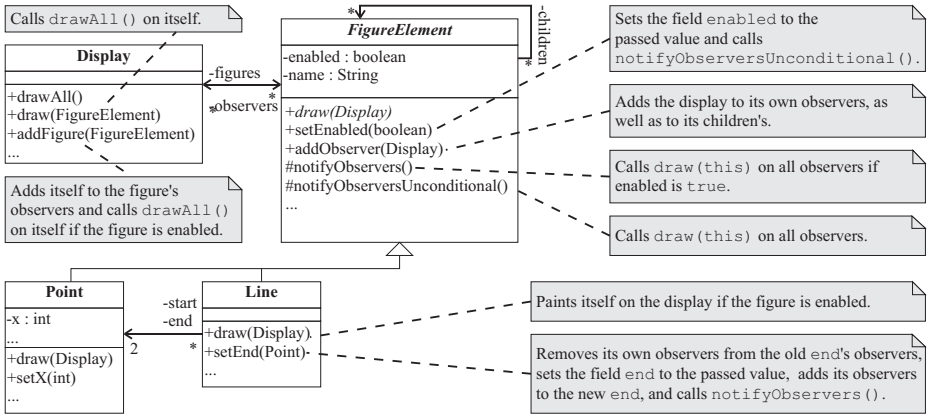


**Fig. 1.** OO implementation of a precise version of the display updating

to the method `Display.addFigure(FigureElement)`; the after-advice associated with this pointcut establishes a subject-observer relation between the receiver and the argument.

The `setSubFigure` pointcut captures points in the execution, where parent-child associations are changed - these are assignments on any field of type `FigureElement` or a subtype thereof (denoted by the "+"), declared in `FigureElement` or any of its subclasses. The before and after advice associated with this pointcut make sure that the observer lists are updated accordingly. The `change()` pointcut captures assignments to those fields of figure element objects that affect the draw behavior - the set of relevant fields includes any field declared in `FigureElement` or one of its subclasses, excluding the fields `FigureElement.name`, `FigureElement.observers` and `FigureElement.children` (the latter two are introduced by the aspect). The advice associated with this pointcut ensures that notifications are sent to relevant observers.

### 2.2 Comparison of the OO and AO solutions

The main advantage of the AspectJ solution over the OO counterpart is that the display updating protocol is made explicit and *localized* in one module. Due to this separation changes to the display update protocol are localized within the aspect. For example, assume that we decide to modify the protocol as follows. The display update signaling currently performed within methods that change the state of figure elements, should happen at caller sites of these methods (e.g., the because the caller object should be logged, which is not possible at execution site). Changes needed to introduce the modified protocol are localized within the aspect code in the AO solution (alternatively a new aspect can be implemented); the same changes are not localized in the OO solution. Furthermore, the separation makes the display updating logic pluggable. The advantages resulting from the separation of crosscutting concerns are discussed elsewhere [22, 16] and are not in the focus of this paper.

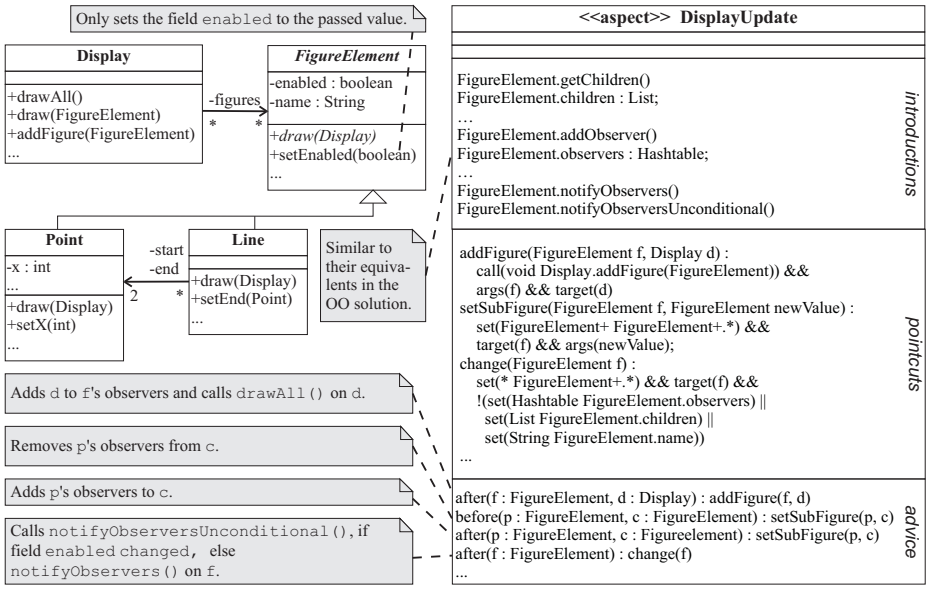


**Fig. 2.** First AO implementation of the precise display updating

Physical separation and localization of concerns, while important, is only one aspect of modularity. Another, equally important aspect of modularity is about the interface that controls the interaction of the separated logic with the rest of the system, thereby employing abstraction mechanisms to hide implementation details. The interface of the separated observer protocol to the rest of the system is defined by pointcuts in the aspect in Fig. 2. A recent paper by Kiczales and Mezini [23] argues that this explicit interface makes modular reasoning in the presence of change easier in an AO setting compared to an OO setting, where there is no explicit interface between these two concerns.

In this paper, we go one step further and investigate the ability of the AO interfaces to absorb change by means of information hiding. Unfortunately, interfaces supported by current pointcut technology fall short in this respect. The set of points in the execution of figure elements where to update appropriate display objects are not defined intentionally by some common semantic property, say, as points where *"changes occur on fields that were previously read in the control flow of the last* `drawAll` *call"*. Rather, the pointcuts in our example mostly describe these points by their syntax, thus, exposing implementation details of the figure element hierarchy to the aspect. The following comparison of the OO and AO solutions from Fig. 1 and Fig. 2 shows that the lack of proper support for information hiding makes the separated display update protocol basically as fragile w.r.t changes as the OO solution. The comparison is organized around the change scenarios presented in Fig. 3, which also summarizes the robustness of the AO and OO solution related to these change scenarios.

First, both scenarios are fragile with respect to scenarios *Ch1* and *Ch3*. In both solutions, moving parts of a figure element's state to a helper class will cause changes

| name | description | example | OO | AO |
|---|---|---|---|---|
| Ch1 | Object graph change: Outsource part of the drawing relevant state of a figure element to a class that is not in the `FigureElement` hierarchy. | Use an object of type `Pair` to store the coordinates of a `Point`. | – | – |
| Ch2 | Class hierarchy change: Inserting a new type into the hierarchy of `FigureElements`. | Adding the class `Circle extends FigureElement`. | +/– | +/– |
| Ch3 | Control flow change: Change the condition under which a display update is necessary. | Renaming the field `enabled` to `visible`, or adding a field `hidden`. | – | – |
| Ch4 | Class definition change: Inserting/removing a field whose change makes display update necessary. | Adding the field `color` to the class `FigureElement`. | – | + |
| Ch5 | Class definition change: Inserting/removing a field whose change does not affect display. | Adding the field `changeHistory` to the class `FigureElement`. | + | – |

**Fig. 3.** Change scenarios with comparison of AO and OO solution

to those fields to escape observation, although they might have had effect on the drawing behavior. Hence, they break w.r.t. *Ch1*. Also, renaming the field `enabled`, or adding a new field which also controls when figure elements are displayed, say `hidden`, will break both the AO and OO protocols. This is because the names of such fields are hard-coded in the implementation of `notifyObservers()`, which is the same in both solutions. Hence, both solution fail to absorb *Ch3*.

The AO solution is more robust w.r.t *Ch4*. The OO protocol breaks in the sense that the display update signaling for the field being added, needs to be adopted, respectively encoded anew. The AO protocol that uses wildcards for pattern matching on names of fields that affect drawing behavior carries over automatically. However, the AO solution is less robust than the OO solution w.r.t *Ch5*. This is because the `change` pointcut in Fig. 2 enumerates each field to exclude from the observation explicitly. Adding (or removing) a field which does not influence the graphical representation of a figure will break field precision of the aspect: Changes to these fields will cause the display to be updated.

Finally, with regard to the scenario *Ch2*, we argue that both solutions perform more or less the same under the assumption that the new class, in general, introduces both fields that affect the drawing behavior as well as fields that do not affect drawing. The AO solution performs better for fields that affect drawing: The protocol established by the aspect automatically applies to the new class. This is not true for the OO solution: The whole logic concerning children and field change should be manually coded in the new class. However, the opposite is true for fields that do not affect drawing and, hence, need to be excluded explicitly in the AO solution.

The use of wildcards for pattern matching on names might at first sight appear to support some sort of abstraction by providing a means to identify relevant execution points by some commonality. However, pattern matching on names only allows to ab-

stract over syntax, which is not always sufficient. Our investigation shows that wildcards do not actually increase the ability to absorb change, beyond simple cases, where there are no exceptions to be made from the rule defined by wildcards. As for our example, we could as well have used an AO solution that does not use wildcards but enumerates the relevant points. This solution would exhibit the same robustness w.r.t the change scenarios as the OO solution[5].

The discussion suggests that without more powerful mechanisms for information hiding the potential of AO mechanisms for improving modularity cannot fully be unleashed. This has been the motivation for us to work on a pointcut language that enables better modularity and information hiding. This language will be presented in the following section. Please note that the question is not whether AO mechanisms provide better modularity than OO mechanisms; the question is rather how to further improve the power of the modularity of AO mechanisms. As mentioned in the beginning of this sub-section, AO does support better modularity by separating and localizing the display update logic [22, 16] and by providing an explicit interface [23]. The point we want to improve is that the focus of pointcuts should be *when* (under which conditions) a pointcut should be triggered rather than *where* (lexically) the corresponding places in the code are.

## 3 The ALPHA language

ALPHA is an AO extension of a toy OO core language implemented as an interpreter in Java. The OO core of ALPHA is based on L2 [12] - a simple object-oriented language in the style of Java. The formal syntax, semantics, and type system of L2 are described in [12]. Here we present the OO core of ALPHA informally by means of the example in Fig. 4 - a simplified variant of the example from the previous section. ALPHA supports classes and single inheritance and has a standard static type system.

### 3.1 Pointcuts and advice

Every class in ALPHA may define fields and methods and may also define pointcuts and associate advice with them. Pointcuts are Prolog queries over a database of both static and dynamic information about the program or program execution. A Prolog query is a sequence of primitive queries combined by the *and* operator "`,`". A simple pointcut that denotes "all assignments to fields of objects of type point" is shown in the class `DisplayUpdate` in Fig. 5.

In contrast to AspectJ and similar to Caesar [29], aspects in ALPHA become effective only after they are *deployed*. Further discussion of this strategy is available at [29]. What matters is to note that the advice of `DisplayUpdate` will have semantic effect once an instance of `DisplayUpdate` is deployed. For illustration, consider the use of the aspect `DisplayUpdate` within the `main()` method of class `Main` in Fig. 5 - here, the advice of `DisplayUpdate` will be effective only during the execution of `doSomething()`.

[5] In [2], the reader can also find code for an AspectJ solution of the example problem that does not uses wildcards.

```
1  class FigureElement extends Object {
2    String name;
3    void draw(Display d) {}
4  }
5  class Point extends FigureElement {
6    int x, y;
7    boolean enabled;
8    void draw(Display d) {
9      if (this.enabled) d.paintPoint(this.x, this.y);
10   }
11 }
12 class Line extds FigureElement {
13   Point start, end;
14   void draw(Display d) {
15     if (this.enabled) d.paintLine(this.start, this.end);
16   }
17 }
18 class Display extends object {
19   FigureElement f1, f2;
20   void drawAll() { this.f1.draw(this); this.f2.draw(this); }
21   void draw(FigureElement fe) { print("display update:"); print(fe); this.drawAll(); }
22   void paintPoint(int x, int y) { ... }
23   void paintLine(Point start, Point end) { ... }
24 }
```

**Fig. 4.** Figure elements in ALPHA

In order to explain the pointcut in Fig. 5, it is necessary to understand the basic structure of the database. The database contains both static and dynamic information about the program organized in a set of relations. A very simple relation is the unary relation `now`, denoted `now/1`. This relation has only one fact that contains the current timestamp. These timestamps are necessary in order to reason about temporal relations between events. The query `now(ID)` in Fig. 5 retrieves the current timestamp and binds it to the variable `ID`. In Prolog, all names starting with uppercase letters are considered variables, whereas all names starting with lowercase letters or enclosed by single quotes (') are considered constants.

```
class DisplayUpdate extends Object {
  Display d;
  after now(ID), set(ID, _, P, _, _), instanceof(P, 'Point') { this.d.draw(P); }
}

class Main extends Object {
  Display d; DisplayUpdate du;
  void main() {
    this.d = new Display(); this.du = new DisplayUpdate();
    this.du.d = this.d;
    deploy(this.du) { this.doSomething() }
  }
  void doSomething() { ... }
  ...
}
```

**Fig. 5.** Simple advice in ALPHA

The second part of the query, `set(ID, _, P, _, _)`, queries a relation `set/5` that stores all assignments in the current execution. The first element of this relation is the timestamp of this event, the second one is a reference into the syntax tree of the program and denotes the expression in the syntax tree that corresponds to this event. The third element is the object that contains the field, the fourth is the fieldname, and the fifth is the value assigned to the field. By using the name `ID` for the first element in the query `set(ID, _, P, _, _)`, we specify that this pointcut will match only if the assignment has happened right now and not in some time in the past because a variable (`ID` in this case) must be bound to the same value in all places where it is used. The wildcard "_" is used for anonymous variables that are not interesting; in the set part of the pointcut in Fig. 5 the "_" wildcards are used to denote that the pointcut matches for any expression as well as for any field name and value being assigned. By using the name `P` for the receiver element of the `set/5` relation, we bind the receiver object of each matching assignment to `P`. In the third part of the pointcut, `instanceof(P, 'Point')`, we constrain the set of eligible assignments even further by requiring `P` to be an instance of the class `Point`.

Variables in a pointcut can be used both as constraints during the unification process and as a means to make context available in the advice. In our example, we use `P` in the advice body[6]. The form of advice is similar to AspectJ [21].

The kind of data in the database is obviously an important variation point of our approach. In our prototype, the database contains four different program models: a representation of the abstract syntax tree, a representation of the object store (heap), a representation of the static type of every expression in the program, and a representation of the trace of the program execution. These four structures are a natural choice, since they represent the main entities used for interpreting a program. However, it would not invalidate our approach to add or remove other entities, e.g., add a model about resource consumption or remove the object store model. In the example above, the `set/5` relation belongs to the execution trace model, whereas the `instanceof/2` relation belongs to the object store model and the static type model. A full reference of the relations in the database is available in the appendix in Fig. 14. The basic idea is that each of these models represents a partial view of the program semantics. By making as much information available to the pointcut programmer as possible, the programmer can choose the program model to be used to express his intention as directly and conveniently as possible.

The escape symbol `@` can be used to evaluate an expression inside a query, such that object-specific constraints involving values from the enclosing object can be expressed. Fig. 6 shows a refined version of the display update aspect whose pointcut will match only if the target of the assignment is `this.p`.

### 3.2 Pointcut abstraction and pointcut libraries

The expressiveness of our pointcut language is due to the rich program models *and* its abstraction mechanisms. Due to Prolog, it is easy to define new predicates that abstract

[6] We use type inference to determine a static type for every variable inside a query, which is used to type-check the advice body. We elaborate on this in Sec. 5.

```
class DisplayUpdate extends Object {
  Display d; Point p;
  after now(ID), set(ID, _, @this.p, _, _) {
    this.d.draw(this.p);
  }
}
```

**Fig. 6.** Inserting context into a pointcut

over the primitive generated predicates. Fig. 7 shows an excerpt of the standard pointcut library building on this feature. Line 2 shows - by the example of the primitive `set` predicate - how to define convenient abbreviations of the generated primitive execution trace predicates for the case that we are not interested in past events or in the syntactic location of the event. With these abbreviations the pointcut in Fig. 6 can be written more conveniently as `set(@this.p,_,_)`.

Lines 6-9 demonstrate the usefulness of having the complete history of execution[7]. The `cflow` query specifies under which conditions a join point `ID0` is or has been in the control flow of another join point `ID1`. Reasonably, this other join point may only be a method call. Thus, `ID1` must be the timestamp of a method call. Method calls are stored in the database as pairs of `calls/5` and `endcall/3` facts denoting the beginning respectively the end of a method call. The `before/2` relation is a part of the execution trace model and can be used to compare events w.r.t. their temporal order. The first rule of the `cflow` query, applies when the `ID1` call has completed (i.e., a corresponding `endcall` fact is available). The second rule applies when `ID1` is still on the call stack; it uses the special control predicate `\+`, which succeeds, if the goal cannot be proven (a.k.a. *negation as failure*).

Please note that this `cflow` construct is much more powerful than the AspectJ pointcut designator with this name, since the AspectJ variant can only be used to refer to control flows that are currently on the call stack (corresponding to our second `cflow` rule), whereas our `cflow` also applies to control flows in the past[8].

The pointcut library contains a set of other pointcut predicates that are only sketched in Fig. 7. We have defined predicates to determine whether an object is reachable from another object following a path of links in the object graph (`reachable/2`) and to determine the class of an object (`instanceof/2`). Other predicates provide convenient access to the AST (`class/2`, `method/3`, `field/3`, `within/3`, `subtypeeq/2`).

The `pcflow/3` predicate predicts the control flow of a method based on the AST, basically building the call graph of the method. This is achieved by computing the transitive hull of all outgoing method calls, whereby all method implementations in all subtypes are considered in order to take late binding into account.

---

[7] Due to the optimizations discussed in sec. 5 only parts of the execution history are recorded that are relevant for the pointcuts in the program.

[8] Note, however, that the main purpose of this paper is *not* to propose new control flow pointcut designators. The `cflow` pointcut designator is just an illustration of the extensibility of our pointcut language.

```
1  % abbrevations if we are only interested in the current event
2  set(Receiver, Field, Val) :- now(ID), set(ID, _, Receiver, Field, Val).
3  % abbreviations for new, calls, get similarly
4
5  % is ID0 in the control flow of ID1?
6  cflow(ID0, ID1) :- calls(ID1, _, _, _, _), before(ID1, ID0),
7                     endcall(ID2, _, ID1, _), before(ID0, ID2).
8  cflow(ID0, ID1) :-
9     calls(ID1, _, _, _, _), before(ID1, ID0), \+ encall(_, _, ID1, _).
10
11 % is Obj2 reachable from Obj1 in the object graph?
12 reachable(Obj1,Obj2) :- ...
13
14 % is Obj an instance of C?
15 instanceof(Obj, C) :- ...
16
17 % convenient access to AST: query classes, methods, and fields
18 class(Name, CDef) :- ...
19 meth(CName, MName, MDef) :- ...
20 field(CName, FName, FDef) :- ...
21
22 % is Expr within method MName of class CName
23 within(Expr, CName, MName,) :- ...
24
25 % is C1 subtype of C2?
26 subtypeeq(C1, C2) :- ...
27
28
29 % is Expr in the statically predicted control flow of CName.MName?
30 pcflow(CName, MName, Expr) :- ...
31
32 % find the most recent event matching a pattern X
33 mostRecent(ID,X) :- ...
```

**Fig. 7.** Excerpts of the pointcut library

The `mostRecent/2` predicate finds the most recent occurence of an event pattern. We will use this predicate to express things like "find the most recent occurence of a call to `draw`".

A very convenient property of Prolog is that these predicates can be used with arbitrary instantiation patterns. This means that the predicates can be used in any direction. For example, the `within/3` predicate can be used to find an expression within a given method and class, or the other way around, to find a class and a method that lexically contain an expression.

The full definition of these predicates can be found in Fig. 15 in the appendix. For the purpose of this work, the details of their definition are not very important. The interesting point is that ALPHA has an *open* pointcut language, whereby new pointcuts can be added on-demand in a *declarative* way. Simple pointcuts can be combined to more powerful pointcuts, so we have the same kind of abstraction mechanism for pointcuts that functional abstraction provides for functions.

We have developed a rudimentary module mechanism for pointcut libraries. Currently, we have a standard pointcut library, that is always available, and user-defined pointcut libraries, that must have the same file name as the source file. A pointcut library can import other libraries using Prolog's own module mechanism. It would be a

straightforward extension to make this a full-fledged module mechanism with explicit imports and exports, namespaces, etc.

## 4 Programming with ALPHA

In this section, we demonstrate how information from different program models can be combined to increase the abstraction level of pointcuts. Furthermore, we discuss the implications of our pointcut language on the programming model.

### 4.1 Expressiveness of pointcuts

The class `DisplayUpdate` in Fig. 8 shows six different ways to specify a display update pointcut in ALPHA, using different models of the program. We use these six differents pointcuts in order to show how we can gradually increase the abstraction level of the pointcuts by exploiting the available information in the database. The resulting pointcuts differ in their support for *robustness* and *precision*, as discussed below and summarized in Fig. 9.

```
1  class DisplayUpdate extends Object {
2    Display d;
3
4    // enum pointcut
5    after set(P, x, _); set(P, y, _); set(P, 'start', _); set(P, 'end', _),
6            instanceof(P, 'FigureElement') { this.d.draw(P); }
7
8    // set* pointcut
9    after set(P, _, _), instanceof(P, 'FigureElement') { this.d.draw(P); }
10
11   // pcflow pointcut
12   after now(ID), set(ID, ExpID1, P, F, _), instanceof(P, 'FigureElement'),
13           pcflow(Display, 'drawAll', (_, get((ExpID2, _), F))),
14           hastype(ExpID2, 'FigureElement') { this.d.draw(P); }
15
16   // cflow pointcut
17   after set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, @this.d,'drawAll', _)),
18           cflow(T1, T2), instanceof(P, 'FigureElement') { this.d.draw(P); }
19
20   // cflowreach pointcut
21   after set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, @this.d,'drawAll', _)),
22           cflow(T1, T2), reachable(Q, P), instanceof(Q, 'FigureElement') { this.d.draw(P); }
23 }
```

**Fig. 8.** Six display update pointcuts

The `enum` pointcut (line 5, Fig. 8) enumerates[9] all assignments to fields that potentially effect drawing behavior, namely to fields `x`, `y`, `start`, or `end` of any object `P` of type `FigureElement`. It uses the names of the fields to identify the relevant assignments. By precisely enumerating the fields potentially involved with drawing, the pointcut supports some sort of static field precision: It makes at least sure that changes

[9] A semicolon denotes "or" in Prolog

to fields that are never read in any control flow of `drawAll()` do not trigger display updates. However, it requires the programmer to explicitly encode this knowledge. Furthermore, it does not take into account the actual control flow of the concrete program execution and, hence, cannot avoid e.g., updates after assignments to fields of disabled points. Also, object precision is not supported. Precision w.r.t. fields involved in the drawing behavior only under certain dynamic conditions - for convenience let us call this dynamic field precision - and object precision require knowledge from dynamic program models, which this pointcut does not make use of. With respect to robustness, the `enum` pointcut exhibits the same behavior as the OO solution in Sec. 2.

The `set*` pointcut (line 9) is triggered by assignments to *any* field of a `FigureElement` object. Due to the use of the _ wildcard this pointcut may cover too many execution points whose signature matches the pattern by accident [15, 20, 13]. As a result, the pointcut performs poorly w.r.t. field precision: Any assignment to the field `name` which is not at all involved with drawing will also trigger a display update. Similar to `enum`, `set*` uses only static information, hence, it supports neither dynamic field precision nor object precision. As far as robustness is concerned, `set*` exhibits the same behavior as the AO solution in Sec. 2.

| criteria | enum | set* | pcflow | cflow | cflowreach |
|---|---|---|---|---|---|
| static field precision | + | - | + | + | + |
| dynamic field precision | - | - | - | + | + |
| object precision | - | - | - | + | + |
| Ch1 | - | - | - | - | + |
| Ch2 | +/- | +/- | + | + | + |
| Ch3 | - | - | + | + | + |
| Ch4 | - | + | + | + | + |
| Ch5 | + | - | + | + | + |

**Fig. 9.** Evaluation of pointcuts w.r.t. change scenarios from Fig. 3

The `pcflow` pointcut (line 12) uses the `pcflow` predicate to approximate the control flow of `Display.drawAll()` based on the AST model and selects field read expressions in the approximated control flow; only assignments to such fields match the `pcflow` pointcut. Similar to `enum`, this pointcut ensures that changes to fields that are never read in the control flow of `drawAll()` do not trigger display updates. However, neither object nor dynamic field precision is supported, since the pointcut only makes use of the AST and not of the dynamic models of the program. The pointcut is not robust in the case of scenario *Ch1* - outsourcing part of figure element state to external objects. The pointcut explicitly requires `P` to be a `FigureElement` in order to be able to pass it to the `draw()` method call. So, state outsourced to non `FigureElement` objects escape the observation by this pointcut.

Note that while supporting the same precision as `enum`, `pcflow` is much more robust. This is due to the abstraction capabilities of the pointcut language (including functional composition and higher-order pointcuts), which allows us to compose primitive

pointcuts into more powerful ones, such as `pcflow`. With a pointcut language that does not support such abstraction mechanisms, e.g., AspectJ's pointcut language that only provides operations on sets - union (`||`), intersection (`&&`), negation (`!`) -, the programmer cannot express the intention to "first identify all field accesses in the control flow of a certain method and than select set operations to these fields" in terms of a generic description, if this functionality is not available as a primitive pointcut designator. (S)he is basically left with the explicit enumeration of such field accesses, as in `enum`; the only alternative to enumeration is to describe general rules by wildcards, which is actually not better with regard to robustness.

The `cflow` pointcut (line 17) is similar to `pcflow` in that this pointcut also selects field reads in the control flow of `drawAll`. The crucial difference is that `cflow` is based on the actual control flow at runtime rather than on a conservative static approximation of it. As a result, `cflow` performs better than `pcflow`. It supports both dynamic and static field precision as well as object precision: Only assignments to a field `F` of an object `P` that are read in the control flow of the particular display object denoted by `this.d` (see the expression `@this.d` in the `pcflow` pointcut) trigger an update. Changes of any field of any figure element that is not referred to by our active display denoted by `this.d` do not trigger updates. By its use of the dynamic execution model of the program, `cflow` significantly improves over `pcflow`. Note that it would not be possible to express something similar with the AspectJ `cflow` construct because the `drawAll` method call is in the past and not on the call stack. The only problem with `cflow` is lack of robustness w.r.t. *Ch1*.

The `cflowreach` pointcut (line 21) solves the robustness problem of `cflow` w.r.t *Ch1*. This pointcut composes the `cflow` pointcut with the `reachable` predicate from Fig. 7/Fig. 15. That is, in addition to assignments to objects of type `FigureElement`, it also captures assignments to any object that is reachable in the object graph from an instance of `FigureElement`. The use of the object graph model makes `cflowreach` robust against *Ch1*. Since it also inherits all features of the `cflow` pointcut, `cflowreach` fulfills the precision requirements and is robust w.r.t. all change scenarios *Ch1* to *Ch5*.

The foregoing analysis demonstrates how our approach enables robust and precise pointcuts. The pointcuts `cflow` and `cflowreach` above encode minimal knowledge about implementation details of the crosscutting structure they describe. They directly express the semantic properties of the display update structure rather than relying on implementation details of how the latter syntactically appears in the program code (the names of the variables involved with drawing are irrelevant for the display update behavior).

This is due to the rich models of program semantics underlying these pointcuts as well as the abstraction mechanisms of the pointcut language. In our approach, the programmer can, however, choose which models of the program (s)he wants to use to express a pointcut: from pure syntactic to very dynamic, operational properties, whichever describe the crosscutting best. In this context, please also note the role of unification in elegantly expressing relations between join points. This is illustrated e.g., by the `cflow` pointcut, where unification together with the `cflow` predicate is crucial in expressing

the temporal relation between points where variables are read respectively written in the execution flow of `drawAll`.

### 4.2 Expressive power, openness, and simplicity

In this section, we reason about the complexity of the programming model of our pointcut language. We argue that in addition to increasing the expressiveness of the language, the rich models of program semantics and the powerful abstraction mechanisms such as Prolog's unification also decrease the complexity of the programming model.

First, consider the version of ALPHA, call it `Fixed-`ALPHA with a fixed pointcut language, including e.g., only the predicates defined in our standard library. The expressiveness of this language is increased as compared to AspectJ - all pointcuts in Fig.8 are written in this language. Nonetheless, the programming model is not more complex than that of AspectJ-like languages [21]. Similar to AspectJ, the programmer needs to understand the meaning of some predefined pointcuts, such as `cflow`, `within`, etc., as well as the semantics of Prolog operators/unification for composing them.

Now let us consider the full ALPHA language, in which (domain-specific) pointcut libraries can be defined as outlined in Sec. 3.2. One may argue that this introduces the complexities of full meta-programming into AOP. Similar to [15], we argue that the problems with full meta-programming occur only in an imperative type of language where the programmer is directly involved with some sort of program transformation.

With ALPHA, the programmer only specifies where and what behavioral effect to apply and is not concerned with how this effect is achieved in terms of operational details. To support our argumentation, two examples are discussed in the following which demonstrate that richer program models and more powerful abstraction mechanisms decrease rather than increase the complexity of the programming model.

First, we review the pointcuts `cflow` and `pcflow` from Fig. 8. They both identify assignments involved in the display update crosscutting by their property of accessing variables previously read in the control flow of `drawAll`. However, the models they use are different. The `cflow` predicate uses a richer model that includes the execution trace; `pcflow`'s model is the AST on top of which it approximates the dynamics. We already argued in Sec. 4.1 that `cflow` specifies the crosscutting structure more precisely. Nonetheless, `cflow` is less complex and easier to understand than `pcflow` (see respective definitions in Fig. 15); The approximation of the dynamics of execution based on the AST model adds accidental complexity to `pcflow`'s definition.

Second, we compare the ALPHA implementation of display updating using the `cflow` pointcut in Fig. 8, with the AspectJ solution shown in Fig. 10. The latter is operationally equivalent to the former: it tries to express the rule *"whenever changes are performed on fields that were previously read in the control flow of the last* `drawAll` *call, make an update"* by quantifying over the dynamic control flow. However, to compensate for the lack of the needed information about the dynamic execution trace, a model of the latter is constructed and managed by the programmer within the aspect. Especially, in lack of more powerful abstraction mechanisms beyond operations on sets, building this model employs the imperative Turing-completeness of Java.

Concretely, the aspect administers observer lists for individual fields rather than for whole objects; an instance field of type `Hashtable` is added into the class `Fig-`

ureElement, whose keys are field names and whose values are the corresponding lists of observers, i.e., Display objects. A display is made an observer of those fields that have been read during the last execution of its drawAll() method (see the after advice associated with the pointcut reads in Fig. 10). The pointcut change captures assignments to fields of figure elements binding the receiver object to f; the after advice associated with it uses f together with the name of the assigned field to retrieve displays that observe the field, if any. Before calling draw on each observer display, the latter is removed from all observer lists it is in, since different fields might get read during the next draw (field precision).

Like the cflow-based solution in ALPHA, the implementation in Fig. 10 is robust w.r.t. all change scenarios (but *Ch1*). However, the aspect schematically shown in Fig. 10 is very complex as compared to the pointcut-advice cflow in Fig. 8. Instead of declaratively defining the crosscutting structure employing functional abstraction, as its ALPHA counterpart does, the aspect employs the imperative Turing-completeness of Java to build up a complex infrastructure to basically reverse-engineer the dynamic execution; trying to make it robust w.r.t *Ch1* will further increase the complexity.

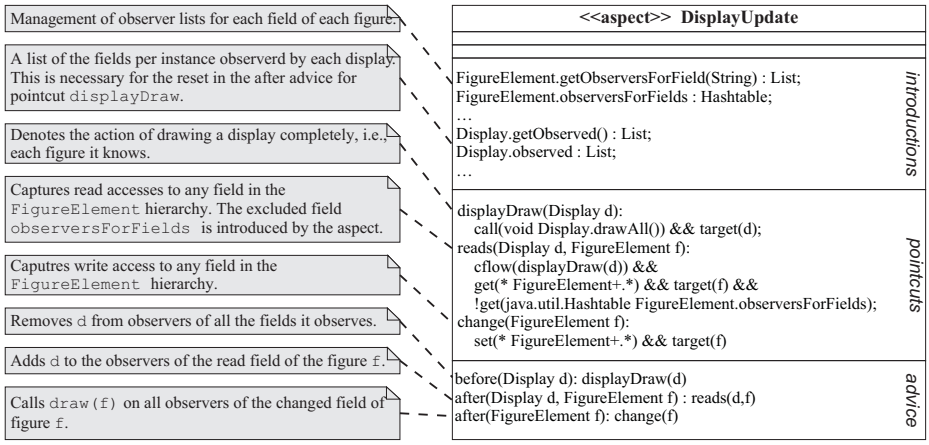


**Fig. 10.** More robust AO implementation of the precise display updating

All the above said on the decreased rather than increased complexity of the programming model, we would like to add that further investigation is still in place to judge whether the Turing completeness of Prolog is actually needed. It would clearly be desirable to have a simpler, but still sufficiently expressive, pointcut language, both for further decreasing the complexity of the programming model as well as for making an efficient implementation of the language simpler. We did not, however, want to restrict the expressiveness of our language from the very beginning and will consider this issue in future investigation.

## 5 Abstract interpretation of pointcuts

A naive implementation of our approach that extends the Prolog database and evaluates all pointcuts after *every* computation step is obviously not acceptable from both *time* and *space* perspectives.

In this section, we present a new static analysis technique that evaluates pointcuts statically in order to compute (a) a (small) set of expressions in the AST (i.e., join point shadows) that will potentially influence the result of a pointcut, and (b) the *lifetime* of facts that are generated at these shadows. The interpreter can take advantage of this information by extending the database and evaluating the pointcuts only if an expression from the aforementioned pre-computed set is evaluated, and by discarding data in the database if its lifetime is over. A side-effect of the static analysis is that it also infers static types for query variables used to type check advice bodies.

Our optimization is based on an abstract interpretation [8] of the pointcuts. Abstract interpretation of a program uses its denotation to make computations in a universe of abstract objects so that the result of an abstract execution gives some information on the actual computation [8].

| **Domain** | **Static abstraction** |
|---|---|
| Time stamps | {`now`,`past`}× Expression IDs |
| Values | Types |
| Execution Trace | Virtual Trace |
| Object Store | Virtual Store |

**Fig. 11.** Runtime domains and their static abstractions

In our case, we approximate the runtime domains shown in Fig. 11. The interpretation is done by a special Prolog interpreter (written in Prolog itself) that evaluates pointcut queries based on our abstract domains and collects data about join point shadows and lifetime during the interpretation.

The virtual trace defines all predicates from the execution trace as rules over the abstract syntax tree and the static type model. For illustration we will only consider the `calls/5` predicate. In a similar way, all other predicates of the execution trace are approximated statically. Their exact definition can be found in the appendix in Fig. 16 and on the project website [2]. The `call/5` predicate is defined as follows:

```
1 calls ((Time,ExprID), ExprID, absval(RecTypeC), MName, absval(ArgTypeC))
2 :-
3   within((ExprID, calls((Rec, _), MName, _)), _, _),
4   stype(Rec, RecType),
5   subtypeeq(RecTypeC, RecType),
6   meth(RecType, MName, meth(_, MName, ArgType, _)),
7   subtypeeq(ArgTypeC, ArgType),
8   addshadow((Time, ExprID)).
```

This rule uses the abstractions defined in Fig. 11 in order to create the virtual trace. Timestamps are represented by a pair `(Time,ExprID)`, whereby `Time` is either the constant `now` or it is unbound. To achieve this, we fix the `now` predicate to the definition

`now((now,_))`. This means that all queries getting the timestamp via the `now/1` predicate will have their timestamp in the abstraction fixed to the constant `now`. All other queries will have an unbound variable in the first position of the timestamp; an unbound variable in the first position denotes a query that might refer to the past.

Instead of values, the execution trace uses types of the form `absval(SomeType)`. All method calls (found in the AST via `within`) imply a corresponding `calls` predicate, whereby the information from the static type system (`stype/2` predicate) is used in order to infer the type of the receiver. Subtyping is taken into account by corresponding `subtypeeq` constraints.

Of particular interest is the `addshadow` part of the rule. This is a special predicate that is intercepted by our static analysis. Whenever an `addshadow` goal is encountered, the interpreter adds the corresponding join point shadow (i.e., `ExprID`) and its lifetime (`Time`) to a list of shadows for the pointcut that is currently analyzed.

The definition of the virtual store is relatively straightforward: It defines the `store` and `classOf` predicate in terms of types instead of values. The situation becomes a bit complicated by taking subtype polymorphism into account. We deal with this by letting `store` range over all possible combinations of types in an object – we ignored performance and favored simplicity in our prototype analysis. The definition of the virtual store is also available in the appendix (Fig. 16).

Our pointcut interpreter is implemented as a meta-interpreter in Prolog. Meta-interpreters are a common technique for abstract interpretation of logic programs [7]. Our meta-interpreter is basically the so-called *vanilla* meta-interpreter [36, Program 17.5] extended by a loop detection mechanism and an additional parameter that collects shadows. In order to invoke the pointcut interpreter we first have to substitute the dynamic values in the pointcut expressions with their static abstraction. By evaluating the pointcuts over the abstracted domains with our pointcut interpreter we basically perform a constant propagation analysis through the control flow of a pointcut. The code of the meta-interpreter is available in the appendix (`shadows` predicate in Fig. 16). We do not want to discuss its implementation here in detail because it uses some very Prolog-specific mechanisms.

```
subtypeeq(D, display ),
shadows(
  ( set(P, F, _), get(T1, _, P, F, _), calls(T2, _, absval(D), draw, _),
    cflow(T1, T2), instanceof(P, figureElement )), ,S)
```

**Fig. 12.** Query for shadows of `cflow` pointcut (line 17, Fig. 8)

For illustrating the abstract interpretation process, the query to compute the shadows for the *cflow* pointcut from line 17, Fig. 8 is shown in Fig. 12. The program expressions inside the pointcut (e.g., the `@this.d` expression in Fig. 8) are replaced by an abstract value that is constrained by its static type via a `subtype` constraint.

The meta-interpreter computes *all* solutions of the query on top of the virtual execution trace and virtual store (thereby collecting shadows triggered by `addshadow` goals). It is important that *all* solutions are computed such that the back-tracking evaluation of

queries covers all possible evaluation scenarios at runtime. The abstract values (i.e., types) returned by the pointcut interpreter are also used to get a bound for the static type of pointcut variables, which is then used to type-check advice bodies.

Fig. 13 illustrates the result of computing the shadows for the aforementioned `cflow` pointcut (Fig. 8) in terms of the code from Fig. 4 and a sample main class. The shadows identified by the pointcut-interpreter are framed in Fig. 13. If the lifetime of the produced facts is `indefinite` (i.e., constant `now` has not been found in the timestamp, the expressions are also underlined, otherwise the lifetime is `immediate`.

For example, the call to `draw` and the field reads are marked as "indefinite lifetime" because they could be relevant as past events in the evaluation of the `get` goal in line 17 of Fig. 8. The lifetime of the field assignments is marked as `immediate` because they are only relevant for this query if they are the current `now` event.

```
1  class Point extends FigureElement {
2    void draw(Display d) {
3      if ( this.enabled ) d. paintPoint ( this.x , this.y );
4    }
5  }
6  class Line extends FigureElement {
7    void draw(Display d) {
8      this .foo( true );
9      if ( this.enabled ) d. paintLine ( this.start , this.end );
10   }
11 }
12 class Main extends Object {
13   ...
14   void writeSomething() {
15     this .p2.y := false ; this .p1.x := true ;
16     this .p1.enabled := false ;
17   }
18   void main() {
19     this .p1 := new Point (); this .p2 := new Point ();
20     this .p2.enabled := true ;
21     this .l1 := new Line (); this .l1. start := this .p1;
22     this .l1 .end := new point ;
23     this .d := new Display (); this .d.f1 := this .l1 ;
24     this .d.f2 := this .p2; this .du = new Displayupdate ();
25     this .du.d := this .d;
26     deploy( this .du) { this.d.drawAll() ; this .writeSomething (); }
27   }
28 }
```

**Fig. 13.** The result of abstract interpretation

### 5.1 Results and limitations

The results of the static analysis are directly used in our interpreter in that Prolog facts/ queries are only evaluated at marked shadows. Also, events for shadows that are marked

with lifetime `immediate` are discarded immediately after the evaluation of the corresponding query. Our interpreter can be run both with and without this optimization. The performance gain depends directly on the relation between marked shadows and unmarked shadows. The example in Fig. 8 runs approximately 4 times faster with the abstract interpretation optimization turned on. In a different example, where the percentage of marked shadows to unmarked shadows is smaller, the program runs 300 times faster. This result is not surprising because extending the database and evaluating queries is very expensive, but it indicates that it is possible to have a very expressive pointcut language that is expensive only if pointcuts are used that cannot be projected on a small set of shadows.

Our analysis technique still has several important limitations, though. First, the analysis itself, as it is presented here, is very slow and would not scale to real systems. It is also hard to guarantee termination of the static analysis in all cases; a typical problem of static analysis by meta-circular interpreters [7]. Our primary goal was to show the feasibility of a static analysis only, so we favored simplicity over performance and completeness. We think that our analysis can be embedded into the conceptual framework described by Codish and Søndergaard [7]. They use a different meta-interpreter, a so-called "bottom-up" interpreter, that has better performance properties and is guaranteed to terminate. This is part of our future work.

Another limitation is in the existence of the `indefinite` lifetime because this means that such facts will never be removed from the database. A more fine-grained analysis that computes lifetimes of the kind "this fact can be removed after some event happened" would be desirable in order to remove this limitation. It is of course easy to construct queries that will inherently require indefinite storage of previous events, but in these cases the static analysis could be used to detect those queries and signal an error if the memory requirements cannot be restricted in a reasonable way.

How do we get from our prototype to an efficient implementation in a compiled language? Besides the limitations mentioned above, our representation of the store is not easy to implement efficiently. A trivial solution is to drop the store model from the pointcut language - there are no conceptual dependencies of our approach on the existence of the store (or any other) model. An alternative would be a database-like organization of the store, which is actually part of our future work.

In order to make the evaluation of the queries itself more efficient, we plan to use partial evaluation techniques such as Logen [26] to reduce dynamic pointcut evaluation to a minimum and to inline the remaining dynamic checks at the computed join point shadows.

## 6 Related work

### 6.1 Pointcut languages

Gybels' and Brichau's proposal [15] is related in several ways. Similar to our approach, they use logic programming and unification for matching pointcuts. The insertion of dynamic context into a pointcut similar to our `@expr` expressions is possible by means of *linguistic symbiosis* [3]. As in our approach, pointcuts can be made reusable by means

of logic rules. The possibility of user-defined pointcut predicates or pointcut libraries is not discussed in [15], but this is no conceptual limitation.

The most important difference to our approach is the data model upon which pointcuts can be expressed. In their approach, the data model consists of a representation of the current join point, syntax tree, and some special object reifying predicates. It is hence not possible to encode queries that refer to the execution history or need access to data from the store. An efficient implementation by computing shadows of pointcuts is also discussed but the addition of the whole execution trace as in our case makes the problem much harder. Other works from the same group [19, 37, 4] also use logic meta-programming but consider only the static syntax of the program as data model.

*LogicAJ* [32] is an extension to AspectJ that uses logic variables and unification instead of wildcards in order to make pointcuts more expressive. The data model upon which the pointcuts operate is unchanged, though.

We have developed an extension of Alpha with which it is possible to refer to future events [24]. Due to several limitations of the implementation, this extension should be seen as an experiment to explore the limits of pointcuts and not as a proposal for a practical programming language.

Walker et al have developed an extension to AspectJ for expressing temporal relations between join points [38]. These temporal relations can be expressed via context-free grammars. The program trace is then "parsed" by an automaton for the grammar. Information about the history of the execution is stored in the state of these automata, which is an effective solution to reduce the amount of data that has to be stored. This approach would not be directly applicable to our model because our pointcut language is more powerful than context-free grammars.

Douence et al have proposed a special pattern matching language for execution traces based on Haskell [11]. Other models besides the execution trace are not covered. Many of the issues presented in this paper (integration into the language, context passing, efficient implementation) are not discussed.

*Josh* [5] is an AspectJ-like language with an extensible pointcut mechanism, built on top of Javassist [6]. Josh does not support declarative pointcut specifications. Rather, new PCDs in Josh are implemented as imperative meta-programs on the abstract syntax tree using the Javassist library. Josh basically suffers from the problems of an imperative meta-programming approach, especially with respect to the composability of the PCDs implemented as meta-programs.

Eichberg et al discussed the usage of the functional query language XQuery as an extensible pointcut language [13]. The data model in this approach is an XML representation of the abstract syntax tree. Due to functional abstraction and the module system of XQuery, it is possible to organize reusable pointcuts in libraries. Other data models or the integration into a programming language are not discussed.

Sakurei et al. [34] propose a design to extend AspectJ with object-specific aspects and pointcuts. Our `deploy` statement can be used with a similar effect as the `associate` statement in this approach. Since runtime values can be used directly in our pointcut language, arbitrary object-specific constraints can be expressed and not just those that are defined in a `perObject` clause. On the other hand, the proposal in [34]

has a more a efficient implementation if many instances of the same object-specific aspect are active simultaneously.

### 6.2 Weaving and static analysis

Hilsdale and Hugunin have described the weaving mechanism in AspectJ [17]. The AspectJ weaver also computes shadows for dynamic pointcuts. However, AspectJ has only a fixed, predefined set of pointcut operators, hence it is easier to compute the set of join point shadows statically. Due to the structure of pointcuts, only certain dynamic checks that look at the class of objects or operate on special stacks (for `cflow`), are required, such that these dynamic checks can be directly woven into the code.

A more semantics-based compilation model, based on a simplified model of AspectJ, can be found in [28]. Using partial evaluation, their model can explain several issues in the compilation processes, including how to find places in program text to insert aspect code and how to remove unnecessary run-time checks. Sereni and de Moor describe a static analysis technique [35] for an even more simplified version of the AspectJ pointcut language that allows a more efficient implementation of some pointcuts than the implementation proposed in [28].

Douence et al presented an analysis technique for detecting interactions between aspects [10]. This is complementary to our static analysis, because we simply assume a global ordering among aspects and concentrate on computing *shadows* of single pointcuts. Nevertheless, our abstract interpretation implies a primitive interaction analysis for free, namely in that it becomes trivial to detect whether two pointcuts have intersecting shadows. However, this is not in the focus of our work.

Codish and Søndergaard [7] describe the usage of meta-interpreters for different abstract interpretations of Prolog code. In contrast to their “bottom-up” approach, we use a conventional top-down meta-interpreter with loop-detection. We are not aware of other works that use abstract interpretation for computing join point shadows.

### 6.3 Aspects and modularity

Lopes et al. [27] motivate and speculate about future ”more naturalistic” referencing mechanisms inspired by natural languages, such e.g., *”those (data) read in previous sentence”*, or even *”in this last operation”*. By means of a simple example they illustrate how referencing mechanisms of current programming languages force programmers to circumscribe their intentions in terms of operational details of the underlying machine. They argue that while pointcuts in AOP languages go one important step further in supporting more powerful referencing they do not go far enough, e.g., in that they lack means of temporal referencing. The prototype we presented in this paper provides a very good basis to experiment with programming models that support more naturalistic referencing mechanisms as those envisaged in [27]. Our prototype can be extended to collect more and different kinds of information about the program to support more ”types of referencing”.

Aldrich [1] proposes module constructs that export pointcuts as part of the module specification. The rationale for this is the lack of modular reasoning if pointcuts depend on implementation details of a module. He shows that the implementation of

such modules can be changed without affecting the consistency of the whole system. On the other hand, this approach is also a serious restriction to the programming model because 1) pointcuts of a module have to be anticipated in its design, 2) the existence of these pointcuts in the interface establishes an implicit coupling to the aspects that use the pointcut, and 3) if pointcuts go across modules (as is inherent for crosscutting concerns), the specification of the pointcut interfaces themselves becomes a crosscutting concern. Our approach also tackles this problem, but with very different means, namely by making the pointcut language more powerful, such that pointcut specifications can be made more robust and less dependent on implementation details. On the other hand, we can give no static guarantees because we cannot enforce implementation-independent pointcuts.

### 6.4 Information engineering in program models

There are also some interesting related works outside the domain of programming language design. Efficient ways to manage and retrieve dynamic data about the execution of a program have been discussed by De Pauw et al [9]. Both the works by Lange and Nakamura [25] and by Richner and Ducasse [33] discuss the design of a static and a dynamic model of the program semantics as well as the use of logic rules to collect and combine information from these models in order to improve program understanding and program visualizations. Abstraction mechanisms to select interesting events in the execution of a program are also used in the domain of *debugging*, for example in the work of Jahier and Ducasse [18]. Reiss and Renieris have developed a framework for processing execution traces by reducing the amount of data as it is collected through mechanisms such as automata or context-free grammars [31]. These techniques may be helpful for us in order to further reduce the amount of collected data.

## 7 Summary and Future Work

In this paper we have presented an analysis which shows that current pointcut languages support localization of crosscutting concerns but have some problems with respect to information hiding. And we have described a new pointcut language in the form of logic queries over different models of the program semantics. Together with the abstraction facilities of logic programming, it becomes possible to raise the abstraction level of pointcuts and hence increase the software quality of aspect-oriented code. We have also presented a static analysis technique that can be the starting point of an efficient implementation.

Our future work will concentrate on the embedding of our pointcut language into a real compiled programming language and on further research in efficient implementation techniques that eliminate the limitations of our current analysis.

## Acknowledgments

We would like to thank Gregor Kiczales, Michael Haupt and Michael Eichberg for comments on drafts of this paper.

This work is partly supported by the European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe).

## References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP'05: European Conference on Object-Oriented Programming*. Springer LNCS, 2005.

[2] Alpha project. http://www.st.informatik.tu-darmstadt.de/pages/projects/alpha/.

[3] J. Brichau, K. Gybels, and R. Wuyts. Towards a linguistic symbiosis of an object-oriented and a logic programming language. In *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2002)*, 2002.

[4] J. Brichau, K. Mens, and K. D. Volder. Building composable aspect-specific languages with logic metaprogramming. In *Generative Programming and Component Engineering (GPCE'02)*. Springer LNCS, 2002.

[5] S. Chiba and K. Nakagawa. Josh: An Open AspectJ-like Language. In *Proceedings of AOSD 2004*, Lancaster, England, 2004. ACM Press.

[6] S. Chiba and M. Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proceedings of GPCE '03*, Lecture Notes in Computer Science, pages 364–376. Springer, 2003.

[7] M. Codish and H. Søndergaard. Meta-circular abstract interpretation in Prolog. In T. Mogensen, D. Schmidt, and I. H. Sudburough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566. Springer, 2002.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*. ACM Press, 1977.

[9] W. De Pauw, D. Kimelman, and J. M. Vlissides. Modeling object-oriented program execution. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 163–182, London, UK, 1994. Springer-Verlag.

[10] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*. Springer-Verlag, 2002.

[11] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*. Springer-Verlag, 2001.

[12] S. Drossoupolou. Lecture notes on the L2 calculus. http://www.doc.ic.ac.uk/~scd/Teaching/L1L2.pdf.

[13] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *Second ASIAN Symposium on Programming Languages and Systems (APLAS)*. LNCS, 2004.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[15] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[16] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings OOPSLA '02. ACM SIGPLAN Notices 37(11)*, pages 161–173. ACM, 2002.

[17] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proc. of AOSD'04*. ACM Press, 2004.

[18] E. Jahier and M. Ducasse. Generic program monitoring by trace analysis. In *Theory and Practice of Logic Programming Journal*, volume 2(4-5). Cambridge University Press, 2002.

[19] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of AOSD'03*. ACM Press, 2003.

[20] G. Kiczales. Keynote talk at AOSD '03, 2003.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings ECOOP'97*, LNCS 1241, pages 220–242. Springer, 1997.

[23] G. Kizcales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings International Conference on Software Engineering (ICSE) '05*. ACM, 2005.

[24] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL) at AOSD'05*, 2005.

[25] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 342–357, New York, NY, USA, 1995. ACM Press.

[26] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. In *Theory and Practice of Logic Programming*, volume 4, pages 139–191, 2004.

[27] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr. Beyond AOP: Toward naturalistic programming. In *Proceedings Onward! Track at OOPSLA'03*, Anaheim, 2003. ACM Press.

[28] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003), LNCS 2622*. Springer, 2003.

[29] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings Conference on Aspect-Oriented Software Development (AOSD) '03*, pages 90–99. ACM, 2003.

[30] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[31] S. P. Reiss and M. Renieris. Encoding program executions. In *International Conference on Software Engineering*, Toronto, Ontario, Canada, 2001. IEEE.

[32] T. Rho and G. Kniesel. Uniform genericity for aspect languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn, Dec 2004.

[33] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, Washington, DC, USA, 1999. IEEE Computer Society.

[34] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proc. of AOSD'04*. ACM Press, 2004.

[35] D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of AOSD'03*. ACM, 2003.

[36] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[37] K. D. Volder and T. D'Hondt. Aspect-Oriented Logic Meta Programming. In *Conf. Meta-Level Architectures and Reflection, LNCS 1616*. Springer, 1999.

[38] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, 2004.

# A Appendix

| Format | Example |
|---|---|
| prog(<br>[class(ClassName, SuperClass,<br>[field(FieldType, FieldName), ...<br>[meth(RetType, MethName, ArgType, Expr), ...],<br>[advice(before, Expr), ...]<br>] ...<br>)<br>where Expr has the form:<br>(ExprID, if(IfExpr, ThenExpr, ElseExpr)<br>(ExprID, get(ReceiverExpr, FieldName)<br>(ExprID, seq(Expr1, Expr2)<br>... | prog(<br>[class(point, figureElement,<br>[field(bool, xx), field(bool, yy),<br>field(bool, enabled)],<br>[meth(bool, draw, display),<br>('9:5', if(('9:13', get(('9:8',this),enabled)),<br>('10:7', '10:15', seq(<br>('10:13', get(('10:8', this), xx)),<br>('10:22', get(('10:17', this), yy)))),<br>('11:10', true) ))],<br>[]%no advice<br>)%class point<br>]%classes<br>). %prog |
| stype(ExprID, Type) | stype('2:26', bool) |
| new(ID, ExprID, ClassName, Obj)<br>calls(ID, ExprID, Receiver, MethodName, Arg)<br>set(ID, ExprID, Receiver, FieldName, Value)<br>get(ID, ExprID, Receiver, FieldName)<br>deploy(ID, ExprID, Obj)<br>endcall(ID, CallID, ReturnValue)<br><br>pred(ID1, ID2) % event ID1 happened<br>% immediately before event ID2<br>before(ID1, ID2) % transitive hull of pred<br>now(ID) % gives the current event ID | calls(3, '53:5', iota1, setP1, iota2)<br>set(4, '66:14', iota1, p1, iota2)<br>endcall(5, 3, false) |
| store(Obj, FieldName, Value)<br>classof(Obj, ClassName) | classof(iota2, point)<br>store(iota2, enabled, false)<br>store(iota2, yy, false)<br>store(iota2, xx, true) |

**Fig. 14.** Format of the four program models (AST, static typing, execution trace, object store) available for pointcuts in Alpha

```
% abbrevations if only interested in current event
new(ClassName, Obj) :-
   now(ID), new(ID, _, ClassName, Obj). *
 calls(Receiver, Method, Arg) :-
   now(ID), calls(ID, _, Receiver, Method, Arg).
 set(Receiver, Field, Val) :-
   now(ID), set(ID, _, Receiver, Field, Val).
 get(Receiver, Field) :-
   now(ID), get(ID, _, Receiver, Field).
deploy(Receiver) :-
   now(ID), deploy(ID, _, deploy(Receiver)).

% is ID0 in the control flow of ID1?
cflow(ID0, ID1) :-
    calls(ID1, _, _, _, _), before(ID1, ID0),
    endcall(ID2, _, ID1, _), before(ID0, ID2).
cflow(ID0, ID1) :-
    calls(ID1, _, _, _, _),   before(ID1, ID0),
   \+ encall(_, _, ID1, _).

% is Obj2 reachable from Obj1?
reachable(Obj1,Obj2) :- reachablevia(Obj1,Obj2,[]).
reachablevia(Obj1,Obj2,_) :- store(Obj1, _, Obj2).
reachablevia(Obj1,Obj2,Via) :-
   store(Obj1, _, Obj3), \+ member(Obj1,Via),
   reachablevia(Obj3, Obj2, [Obj3|Via]).

% convenient access of AST
class(Name, CDef) :-
   prog(CDefs), member(CDef, CDefs),
   CDef = class(Name, _, _, _, _).
meth(CName, MName, MDef) :-
   class(CName, class(_, _, _, MDefs, _)),
   member(MDef, MDefs), MDef = meth(_,
MName, _, _).
field(CName, FName, FDef) :-
   class(CName, class(_, _, FDefs, _, _)),
   member(FDef, FDefs), FDef = field(_, FName).

within((ExprID, Expr), CName, MName,) :-
   meth(CName, MName, meth(_, _, _, Body)),
   subExpr(Body, (ExprID, Expr)).
subExpr(E, E).
subExpr(X, E) :-
   X =.. [_|List], member(E1, List), subExpr(E1, E).

% static subtype/subclass relation
directsubtype(C1, C2) :-
   class(C1, class(_, C2, _, _, _)).
subtypeeq(bool, bool).
subtypeeq(C, C) :- class(C, _).
subtypeeq(C1, C2) :-
    directsubtype(C1, C3), subtypeeq(C3, C2).

% add subtyping to static and dynamic types
hastype(ExprID, C) :-
   stype(ExprID, D), subtypeeq(D, C).
instanceof(Obj, C) :-
   classof(Obj, D), subtypeeq(D, C).

%predicted control flow
pcflow(CName, MName, E) :-
   pcflow1(CName, MName, E, []).
pcflow1(CName, MName, E, _) :-
   within(E, CName, MName).
pcflow1(CName, MName, E, V) :-
   within((_, calls((RecID, _), MName1, _)), CName, MName),
   stype(RecID, CName2),
   (subtypeeq(CName1, CName2); subtypeeq(CName2, CName1)),
   meth(CName1, MName1,_),
   \+ member((CName1, MName1), V),
   pcflow1(CName1, MName1, E, [(CName1,MName1)|V]). *

%finding the most recent of an event pattern X
mostRecent(ID,X) :-
   bagof(ID,X,IDs), maxlist(IDs,ID).
```

**Fig. 15.** Standard pointcut library

```
% virtual store
store(absval(CName), Field, absval(TypeC)) :-
        subtypeeq(CName, CSuper),
         field(CSuper, Field, field(Type, _)),
        subtypeeq(TypeC, Type).
classof(absval(CName),CName).

% virtual event trace
calls((Time,ExprID), ExprID,
   absval(RecTypeC), MName, absval(ArgTypeC))
:-
   within((ExprID, calls((Rec, _), MName, _)), _, _),
   stype(Rec, RecType),
   subtypeeq(RecTypeC, RecType),
   meth(RecType, MName, meth(_, MName, ArgType, _)),
   subtypeeq(ArgTypeC, ArgType),
   addshadow((Time, ExprID)).
% similarly for set, get, new, deploy, endcall

now(now).
before(_, _).
pred(_, _).
% the meta-interpreter
shadows(true, [], _) :- !.
shadows((A,B), Shadows, Trail) :-
   !, shadows(A, S1, Trail), shadows(B, S2, Trail),
   append(S1, S2, Shadows).
shadows(X, [], _) :-
    predicate_property(X, built_in), !, X.
shadows(addshadow((Time,Token)), S, _) :-
   var(Time), !, S = [(Token, indefinite)].
shadows(addshadow((now,Token)), S, _) :-
   !, S = [(Token, immediate)].
shadows(A, S, Trail) :-
   loop_detect(A, Trail), !.
shadows(A, S0, Trail) :-
   clause(A, B), shadows(B, S0, [A|Trail]).
```

**Fig. 16.** Meta-interpreter for computing pointcut shadows