Pointcuts as Functional Queries

Michael Eichberg, Mira Mezini, and Klaus Ostermann

Software Modularity Lab, Department of Computer Science
Darmstadt Universtiy of Technology, Germany
{eichberg,mezini,ostermann}@informatik.tu-darmstadt.de

Abstract. Most aspect-oriented languages provide only a fixed, built-in set of pointcut designators whose denotation is only described informally. As a consequence, these languages do not provide operations to manipulate or reason about pointcuts beyond weaving. In this paper, we investigate the usage of the functional query language XQuery for the specification of pointcuts. Due to its abstraction and module facilities, XQuery enables powerful composition and reusability mechanisms for pointcuts.

1 Introduction

Join points and pointcuts are pivotal concepts of aspect-oriented programming (AOP for short). *Join points* are points in the code (static join point) and/or execution (dynamic join point) of a program. A *pointcut* is a set of join points that share common properties, e.g., the set of all execution points of a certain program, where one would like to control access rights. Pointcuts are defined by means of *pointcut designators* - predicates on join points. Once a pointcut is specified, semantic effect at the referenced join points can be defined in a uniform way, e.g., implementing a certain access control policy.

AspectJ-like languages come with a set of predefined pointcut designators, such as e.g., `call` and `get`, that are used as predicates over join points. One disadvantage of this approach is that there is no general-purpose mechanism in AspectJ to relate different join points, only some special-purpose predicates such as `cflow` allow pointcuts to go beyond a single join point. To convey an intuition of this limitation, let us consider identifying all join points where the value of a variable is changed that is previously read in the control flow of a method `display`, the goal being that we would like to recall `display` at any such point. Assuming a hypothetical AspectJ compiler that employs some static analysis techniques to predict control flows, one can write a pointcut `p1` that selects all getters in the predicted control flow of `display`. However, it is not possible to combine `p1` with another pointcut `p2` which takes the result of `p1` as a parameter, retrieves the names of the variables read in the join points selected by `p1`, and then selects the set of join points where one of these variables is changed. What we need is the ability to reason about join points in `p1` and `p2` *simultaneously.*

In this paper we argue that a pointcut language should have a general purpose mechanism to define predicates that relate different join points. In order to show

the value and the feasibility of such AO languages we have implemented an AOP model in which pointcuts are sets of nodes in a tree representation of the program's modular structure, and such sets are selected by queries on node attributes written in a query language and can be passed around to other query functions as parameters.

These concepts are exemplified in the Java context, as follows. We have created an XML-to-class file assembler/disassembler that can be used to create an XML representation of a class file and convert an XML file back into a class file on the basis of our BAT [1] bytecode framework. On top of this XML representation of the program structure, we use XQuery, a standard functional XML query language as our pointcut language. The choice for XML and XQuery is not conceptual, though; the decision was mainly a matter of reusing existing tools - it would have also been possible to define the query language directly on class files but then we could not reuse existing XQuery implementations.

Pointcuts specified as functional queries over some representation of a program have three main benefits. First, queries enable to write *precise specifications of pointcuts*. In current languages, pointcuts are only described informally and have a complicated, imperative implementation. Formal specifications of AspectJ-like pointcut languages exist but the implementation of the poincuts is separated from their specification. On the contrary, queries allow a short and precise specification of the meaning of a pointcut construct (assuming that the semantics of the XQuery primitives themselves is clear – in the case of XQuery, this is backed up by the existence of a formal semantics [20]). Second, poincuts as functional queries enable *open pointcut languages* in a very natural way. By means of our query language users can extend the pointcut language with their own pointcuts: It becomes possible to create libraries of domain-specific pointcuts, e.g., for synchronization, or for optimizations.

Last but not least, a pointcut query language allows to create more *semantic pointcut mechanisms* [6, 10]. Since we consider the support for more semantic pointcuts an important goal of our work, let us clarify what we mean by a semantic pointcut mechanism. We will use for this purpose the display updating example, basically an instantiation of the observer pattern [5], used as a canonical example for AspectJ [10]. In the example, graphical objects of type `FigureElement`, such as `Point`s and `Line`s are shown on a singleton `Display` object (the classes involved are shown in Fig. 1). Furthermore, it is required that any change on the part of the state of the graphical objects which is read during the execution of `FigureElement.draw()` should trigger an update of the display object by calling the method `Display.update()`.

An example implementation of the display updating functionality in AspectJ is shown in Listing 1.1. The aspect modularizes the decisions about (a) where to trigger the update of the display object, as well as, (b) how the triggering should be performed. The "how" is to call `Display.update()` after any element of the "where" set. The "where" consists of invocations of methods whose name starts with `set` declared in `FigureElement+` ("`+`" denotes subclasses `FigureElement`), or of the method `FigureElement.moveBy(int,int)`.

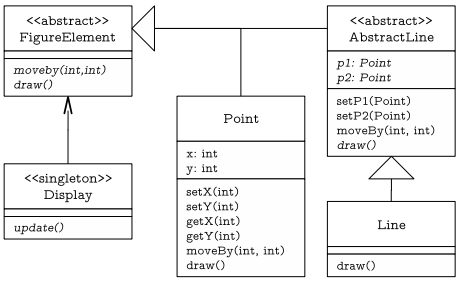

Fig. 1. UML diagram of the FigureElement example

While nicely modularizing the decision about where to trigger the display updating functionality[1], the pointcut in Listing 1.1 is problematic in terms of modular composition of the aspect and the graphical objects [6, 10]. Instead of expressing our intention to *"select points in the execution that modify variables previously read within the control flow of the method* `FigureElement+.draw()`*"*, the pointcut actually relies on implementation details of how the interesting points actually appear in the program code. The problem with such a specification is that it makes the pointcut fragile w.r.t. changes in graphical object classes. E.g., if one adds a new field that does not have a setter method, but is actually read in the control flow of any method `FigureElement+.draw`, changes to this field would escape the aspect, if the latter is not accordingly modified.

Listing 1.1. Display Updating in AspectJ

```
1 after(): call(void FigureElement+.set*(..)) ||
2         call(void FigureElement.moveBy(int, int)) { Display.update(); }
```

To convey the intuition behind more semantic pointcut mechanisms Kiczales [10] gives the example of the `pcflow` pseudo-pointcut[2] shown in Listing 1.2. The pointcut is meant to say: *" (a) predict the control flow of* `FigureElement+.draw()` *and find field reading execution points within it, (b) retrieve the set of the fields being read (denoted by <*`displayState()`*>), and (c) trigger a display update at any execution point where a field contained in <*`displayState()`*> is modified"*. This specification describes the set of the join points we want to select by their semantics - it describes "what" the interesting points are, rather than "how" they are implemented. We call this an *implementation-shy* pointcut[3]. As such, it remains stable toward changes in the implementation mentioned above. One can conclude that implementation-shy pointcut mechanisms make AOP more useful, more principled, more robust.

[1] In an object-oriented solution, the programmer would have to spread the code for triggering the update of the display around the classes `Point` and `Line`.

[2] The "*" after the keyword `pointcut` stands for "pseudo".

[3] This is in analogy to the notion of structure-shy behavior supported by traversal strategies in Demeter [14].

Listing 1.2. PCFlow Pseudo-Code

```
1 pointcut* displayState(): pcflow(execution(void FigureElement+.draw()))
2                           && get(* FigureElement+.*);
3 after set(<displayState()>)(): { Display.update(); }
```

Unfortunately, implementation-shy pointcuts are not properly supported by AOP languages so far. As we already mentioned, the key problem is that with current technology there is no general-purpose support to relate different join points. However, the pseudo-notation <`displayState()`> stands for a reification of the result produced by the `displayState` pointcut, so that the names of the accessed fields can be retrieved and passed as a parameter to the `set` pointcut. We will demonstrate how pointcuts as the one in Listing 1.2 can be expressed as queries.

In this paper, we focus only on static pointcuts, i.e., pointcuts that correspond directly to locations in the source-code/byte-code, also called *join point shadows* in the terminology introduced in [15]. AspectJ's dynamic pointcuts such as `target`, `this`, and `cflow` are not in the focus of this paper. The reason is that these pointcuts do not by themselves define new shadows and are implemented by the AspectJ compiler by inserting conditional logic at shadows selected by static pointcuts [7]. Our focus on static pointcuts is due to the use of the static structure of the program as the data over which to run queries. Also, one of the first usages of this pointcut language is in the context of our XIRC tool [4], whose purpose is to visualize crosscutting structure in the code. However, we think that the notion of pointcuts as functional queries can be applied to any representation of a program, e.g., a representation of the dynamic control flow. This would allow to express dynamic join points more elegantly but is also challenging to implement efficiently. The generalization of our approach to queries on the dynamic call graph is actually the next step in our future work.

The reminder of this paper is structured as follows. In Sec. 2 we give an overview of our representation of class files as XML trees. In Sec. 3, we give a short introduction to XQuery and show how basic and advanced pointcuts can be implemented as queries. In addition, we present first performance results of our implementation, indicating that an efficient implementation is feasible. Sec. 4 presents related work. Sec. 5 concludes.

2 Data Model

The data model on top of which pointcut queries are formulated is closely related, although not identical (see below), to Java bytecode. As mentioned before, all pointcut queries operate on an XML representation of a class file. This XML representation is generated by analyzing the bytecode of a class and can always be transformed back into bytecode. The `BAT2XML` tool is used for this purpose[4]. Representing a class file as an XML document, i.e., as a tree structure, is a very natural thing to do: A class defines methods and fields and each method in turn

[4] The tool can be downloaded from: http://www.st.informatik.tu-darmstadt.de/BAT

defines its functionality by an ordered sequence of (bytecode) instructions. If necessary, sub-sequences of instructions can be further grouped.

The meta-structure of the generated XML representation is shown in Fig. 2. A top-level `all` node represents a program space for which we want to determine shadows corresponding to a pointcut designator[5]. All classes belonging to the program space at hand are represented by the children of `all`. The children of a class node, in turn, define the inherited classes and interfaces as well as the declared fields and methods. While a class node can have many `field` and `method` children it can have at most one `inherits` child node. The `inherits` node can have one `class` node and an arbitrary number of `interface` children. A `field` node basically defines the `name` and the `type` of a field. A `method` node represents a method declaration, consisting of two children, the node representing the `signature` of the method and the node representing its `code`. If the method is native or abstract the `code` child does not exist. However, if it exists, it has one to many children representing the `instructions` of the method.

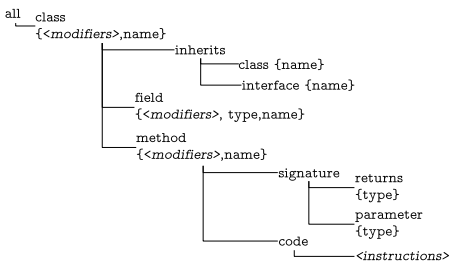

Fig. 2. The meta structure of the XML representation of a program.

The `<modifiers>` of a class, method or field are represented by attributes with boolean values, i.e., if a class is abstract the corresponding node has an attribute `abstract` with the boolean value `true`. The visibility modifiers (`public`, `protected` and `private`) are represented by the `visibility` attribute the value of which is the name of the visibility identifier. E.g., the XML representation of the field definition `private final int length` would be:

```
1 <field visibility="private" final="true" type="int" name="length"/>
```

Our XML representation abstracts over some details of Java bytecode to make the XML representation accessible to a query writer at a higher-level of abstraction. For instance, a field read access is represented in Java bytecode either as a `getfield` or as a `getstatic` instruction, depending on whether the field is static or not. In the XML representation both bytecode instructions are uniformly represented by a `get` node with an attribute indicating whether or

[5] Several strategies similar to those employed in AspectJ, (e.g., `.lst` files) can be used to determine this scope.

not the field is static. The `declaringClassName` and the `fieldName` are the attributes of every `get` or `put` instruction, as shown below:

```
<get declaringClassName="AbstractLine" fieldName="p1"
     static="false" type="Point" />
```

In a similar way, there are different invocation operations in Java byte-code for virtual and static methods. All `invoke` bytecode instructions are represented by one `invoke` node. The attributes of an `invoke` instruction are the name of the invoked method, the name of the declaring class, and the `signature` of the method including its return type, as shown below:

```
<invoke declaringClassName="FigureElement" methodName="moveBy">
  <signature>
    <returns type="void"/><parameter type="int"/><parameter type="int"/>
  </signature>
</invoke>
```

All in all, the XML presentation of the byte code is roughly equivalent to an abstract syntax tree of the source code, modulo some peculiarities of the byte-code format[6].

Note that the introduced abstraction does not lead to loss of expressiveness in writing queries. For most queries (especially queries to express AspectJ pointcuts) the only information we need is that a certain field is read/written by a `get/put` instruction, or that a method is invoked. The abstraction brings our data model a little closer to the structure of the source code. Nevertheless, we do not fully abstract from bytecode details, e.g., there are no explicit `get, invoke` instructions in source code.

For illustration, listing 1.3 shows the XML representation generated for the following Java source code:

```
abstract class AbstractLine extends FigureElement {
  Point p1 = new Point(); Point p2 = new Point();
  FigureElement(){ super(); }
  void setP1(Point point) {this.p1 = point;}
  void setP2(Point point) {this.p2 = point;}
  void moveBy(int x,int y){
    this.p1.moveBy(x, y); this.p2.moveBy(x, y);
  }
  abstract void draw();
}
```

Listing 1.3. XML Representation of the Java class `AbstractLine`.

```
<class abstract="true" name="AbstractLine">
  <inherits><class name="FigureElement"/></inherits>
  <field type="Point" name="p1"/>
  <field type="Point" name="p2"/>
  <method name="<init>">
    <signature><returns type="void"/></signature>
    <code>...</code>
  </method>
  <method name="setP1">
    <signature> <returns type="void"/>
      <parameter type="Point"/>
    </signature>
    <code>
```

[6] For example, exception handling in byte code can in general not be mapped back to the original *try-catch-finally* form

```
14      <load index="0"/>
15      <load index="1"/>
16      <put declaringClassName="AbstractLine" fieldName="p1"/>
17      <return/>
18    </code>
19  </method>
20  <method name="setP2">... as "setP1" but setting p2 </method>
21  <method name="moveBy">
22    <signature> <returns type="void"/>
23      <parameter type="int"/><parameter type="int"/>
24    </signature>
25    <code>...</code>
26  </method>
27  <method abstract="true" name="draw">
28    <signature><returns type="void"/></signature>
29  </method>
30 </class>
```

The XML representation defines the same methods (line 9,20,21,27) and fields (line 3,4) as the source code. The constructor is represented by the method `<init>` (line 5) with return type `void` (line 6). The code of the constructor (line 7,27) and the `moveBy` methods are omitted for brevity. To understand the representation of the `setP1` method it is important to know that the Java Virtual Machine (JVM) is a stack machine. The instruction `<load index="0"/>` (line 14) puts the value stored in the local variable with index 0 onto the stack. This value is `this` and is always stored in the local variable with index 0 at the beginning of an instance method. The parameters passed to a method are stored in the following local variables. In this case, the instance of `Point` that will be assigned to the field `p1` is stored in the local variable with index 1 (line 15) and is also put onto the stack. The `put` instruction (line 16) pops both values from the stack and assigns the top value to the field `p1` of the instance referenced by the 2nd top-most value. Finally, the return instruction terminates the method.

3 Queries

We will first introduce XQuery. Subsequently, we show how AspectJ's primitive pointcuts, such as `execution, call, and within`, as well as pattern matching on signatures can be expressed in XQuery. The goal is to give the reader an intuition of the pointcuts-as-queries metaphor by the relation to something known. The remaining static pointcuts of AspectJ, such as `staticinitialization`, `set`, and `get`, can be expressed in a similar way, and will not be discussed for brevity reasons. Finally, we will demonstrate the power of our approach by a more sophisticated pointcut example and discuss weaving and performance issues.

3.1 XQuery

XQuery [19] is a query language for Extensible Markup Language (XML) data sources. While XQuery is a fully functional language comprised of several kinds of expressions that can be nested and composed with full generality, we will only elaborate on the parts that are relevant to the purpose of this paper. For our purposes, the most important part of XQuery is the notion of *path expressions*.

In a nutshell, a path expression selects nodes in a (XML-)tree. For example, the path expression `$all/class/method` selects all method nodes of the tree corresponding to the XML document from listing 1.3. In general, a path expression consists of a series of *steps*, separated by the slash character. The path expression above has two steps: the *child* steps `class` and `method`. The result of each path expression is a sequence of nodes; in this case all `method` nodes from Listing 1.3.

XQuery supports different directions in navigating through a tree, called *axes*. In the path expression above we have seen the *child* axis. Other axes relevant for this paper are the *descendant axis* (denoted by "`//`"), the *parent axis* (denoted by "`..`"), and the *attribute axis* (denoted by "`@`"). Using the descendants axis rather than the child axis means that one step may traverse multiple levels of the hierarchy. For example, the above query could be rewritten as `$all//method`. The attribute axis selects an attribute of the given context node, whereas the parent axis selects the parent of a given node. For example, the path expression `$all//code/../@name` selects all `name` attributes of all method nodes that have a `code` child, i.e., which are not abstract or native methods. Another important feature of XQuery is its notion of *predicates* - (boolean) expressions enclosed in square brackets, used to filter a sequence of values. For example, the query `$all//method[@name="setP1"]` selects all methods whose name is `setP1`.

One can bind the result of an expression to a variable by means of a `let` expression. Variables in XQuery are marked with the `$` character. As already mentioned, we will use the variable name `$all` for the root element node containing all classes on which we want the queries to operate. XQuery also offers a number of set operators to combine sequences of nodes, namely `union`, `intersect`, and `except`, with the usual set-theoretic denotation, except that the result is again a sequence in document order. The last important feature of XQuery used in this paper is its notion of a function definition. The following function definition subtracts the results of two pointcuts passed to it as parameters. Note that all selection operations in XQuery work on *sequences* of selected nodes: `$p1, $p2` being of type `element()*` means that they are *sequences* of selected elements. Hence, it is possible to pass the result of another pointcut query to this function.

```
1 declare function diff($p1 as element()*,$p2 as element()*) as element()*
2 { $p1/.. except $p2/.. }
```

3.2 Method Execution and Pattern Matching

Let us start with the `execution` pointcut designator (PCD) in AspectJ. As an example, consider finding executions of the method `void setX(int i)` declared in class `Point`. In AspectJ, this would be expressed by the PCD `execution(* Point.setX(..))`. The same semantics can be expressed in our approach by the query `$all/class[@name="Point"]/method[@name="setX"]`. We start with the set of all nodes (`$all`), search from there for class nodes with the `name` attribute `Point` (recall that the `@` operator selects attributes) and select direct method sub-nodes of the latter with the `name` attribute equal to `setX`.

Note how AspectJ's notion of wild cards corresponds to specifying or omitting additional query constraints. E.g., to find all methods named `setX` in *all*

classes (corresponding to `execution(* *.setX(..))` in AspectJ), we would write: `$all//method[@name="setX"]`. Constraints on the signature can be expressed by appropriate conditions. For example, the query
`$all//method/signature/parameter[1][@type="int"]/../..`
selects methods whose first parameter is of type `int`. Note that `/../..` selects the ancestor method node of the `int` parameter node at hand. Similarly, one can select based on return types or modifiers.

XQuery provides a rich library of functions for pattern matching on names. E.g., `$all/class[@name="Point"])/method[starts-with(@name,"set")]` selects all methods whose name starts with `set` in class `Point` (corresponding to `execution( * Point.set*(..))` in AspectJ), whereby `starts-with` is a library function of XQuery. Similarly, other name patterns can be expressed by appropriate calls to these library functions.

3.3 Method Calls and Subtype Predicates

Another important category of pointcuts are method calls, corresponding to AspectJ's `call` PCDs. For such slightly more sophisticated pointcuts, it makes sense to define a reusable XQuery function. This is illustrated by the function `call` below, which given a set of nodes from which to select, denoted by `$all`, and a set of (previously selected) method nodes, `$meths`, passed to it as parameters, selects from `$all` any call instruction to one of the methods in `$meths`. The `=` operator in XQuery is implicitly existentially quantified if it is used on sets/sequences of values: If *there exists* a node in the set `$meths` whose name is the same as `@methodName` the condition evaluates to *true*.

```
declare function call($all element(), $meths as element()*) as element()* {
  $all//invoke[(@methodName = $meths/@name) and
               (@declaringClassName = $meths/../@name)]
}
```

Given the definition of `call`, the following expression selects all calls to any method whose first parameter is of type `int` (corresponding to the PCD `call(* *.*(int, ..))` in AspectJ). This usage of the `call` query function demonstrates the idea of relating different join points: The query parameter passed to the `call` function is itself a pointcut.

```
call($all,$all//method/signature/parameter[1][@type="int"]/../..)
```

An important operation in the context of method calls is the ability to reason about all subtypes of a given type. This is important for expressing predicates of the kind *"calls to a method `m` of class `C` or any of its subclasses"*, corresponding to PCDs of the form `call(* *.C+.m(..))` in AspectJ. All subtypes of a given set `$types` of previously selected classes can be retrieved by the recursive function definition `subtypes` below. This function computes the direct subtypes of `$types` in `$s`. If `$s` is empty, then `$types` is already the result, otherwise the result is the union of `$types` with the result of the recursive call `subtypes($all, $s)`.

```
declare function subtypes($all as element()*, $types as element()*)
    as element()* {
  let $s := $all/class[./inherits//@name = $types/@name]
  return if (empty($s)) then $types else $types union subtypes($all, $s)
}
```

For illustration, the function `subtypes` is used in the following query to select calls to methods in class `FigureElement` or any of its subclasses, whose name starts with the string `set` (PCD `call(* FigureElement+.set*(..))` in AspectJ).

```
call($all,subtypes($all,$all/class[@name="FigureElement"])
                  /method[starts-with(@name,"set")])
```

3.4 Lexical Restrictions and PCD Composition

Lexical predicates on join points such as "*within the code of class X*", expressed by the primitive pointcut `within` in AspectJ, can be expressed in two different ways as queries, which we will illustrate by AspectJ's PCD `within(Line) && get(* *.p1)`. One way is to select by the name of the unit (class or method) serving as the lexical scope. E.g., `$all//class[@name="Line"]//get[@name="p1"]` selects all field read instructions accessing a field `p1` inside the code of the class `Line`. The same PCD can be expressed by `$all//class[@name="Line"]//* intersect $all//get[@name="p1"]`, using the path selection operation `//*`, which selects all sub-nodes of a given node.

Note the use of XQuery's set operation `intersect` above. Together with the path selection operation, it enables to combine lexical restrictions with any other, arbitrary sophisticated, query. For illustration, consider the following query, which corresponds to AspectJ's PCD `within(Line) && call(* Point.getX(.. ))`. The same semantics would be hard to express by the approach to lexical restrictions based on predicates on the name of the lexical element.

```
let $c := call($all,$all//class[@name="Point"]/method[@name="getX"])
$all//class[@name="Line"]//* intersect $c
```

The example just discussed brings us to the issue of composing queries. The logical PCD composition operations from AspectJ, *or* (`||`), *and* (`&&`) and *not* (`!`) can be very naturally expressed in our approach by the corresponding set operators of XQuery, as shown below.

```
pc1 && pc2   <-->   pc1 intersect pc2
pc1 || pc2   <-->   pc1 union pc2
!pc1         <-->   $all except pc1
```

3.5 Advanced Pointcuts

In the introduction, we discussed the need to support more abstract pointcuts and discussed a pointcut designator proposed by Kiczales [10] that predicts the control flow of methods. We presented such a semantic-based pointcut in the introduction and argued that in order to support it, support for relating different join points is needed. In the following we show how the desired semantic can be expressed by pointcut queries.

A very simple (and not very precise) mechanism of predicting the control flow of a set of methods `$m` is specified by the following query. The local variable `$pcflow1` selects all methods that are called inside `$m`. If this set is empty, then `$m`

is already the result, otherwise we compute the next level of methods that are not yet contained in the predicted control flow. The condition `except $m//method` guarantees that the function terminates in the presence of cyclic call structures.

```
declare function pcflow($all as element()*, $m as element()*)
   as element()* {
  let $pcfl1 := $all//method[@name = $m//invoke/@method] except $m//method
  return if (empty($pcfl1)) then $m else pcflow($all, $m union $pcfl1)
}
```

This algorithm for predicting the control flow is not very precise because it considers only method names and not subtyping restrictions or control flow/data flow analysis inside method bodies. Of course, we could define more sophisticated predictions using control- and dataflow techniques, but this is not in the scope of this paper[7]. The purpose of the example is primarily to illustrate the generality and expressiveness of our queries. To serve this purpose, let us take a look at how this function can be used in our observer example focusing on how it can be used to make the pointcut specification more robust. The following query selects all assignment instructions to fields that are read in the predicted control flow of the method `FigureElement+.draw()`, thereby expressing the `pcflow` pseudo PCD from the introduction.

```
$all//put[@name = pcflow($all,
                  subtypes($all,$all/class[@name="FigureElement"])
                           /method[@name="draw"])//get/@name ]
```

Note how an aspect that defines an advice (e.g., to call `Display.update()`) with this pointcut as a parameter abstracts over the control flow between `Figure-Element` and `Display` objects. The advice would be "control flow shy", which means that it will not be affected by changes in the implementation of the base software other than changing the name of the class `FigureElement`, or of the methods `draw` and `update`, hence enhancing modular reasoning supported by aspect modularity.

3.6 Weaving

The language to specify pointcuts is largely independent from the question on how to use a pointcut. It can be used for different purposes, e.g., weaving of advice code, generation of error/warning messages, visualization, support for refactoring etc. In order to create a complete system we have implemented a primitive weaving engine on top of our pointcut language that weaves calls to a central aspect dispatcher before and after every join point of a query result. At runtime, advices can be registered for a pointcut via a corresponding API. Context information (caller, receiver, arguments etc.) can be received via API calls. It would be straightforward to integrate advices like in AspectJ directly into a programming language on top of this API.

[7] We have implemented a more precise version of `pcflow` that takes subtyping into account but we do not present it here due to space reasons.

3.7 Performance

Query	Time in sec.

`let $field := //field[@visibility="protected"]`
`return //get[@declaringClassName = $field/../@name`
`  and @fieldName = $field/@name]`
 	
*get(protected *.*)*
	0.81
	(273 nodes)
	
	2.15

`//invoke[@declaringClassName="de.tud.BytecodePointcut"`
`  and @methodName="getFilter"]`
 	
call(de.tud.BytecodePointcut.getFilter(..))
	0.45
	(5 nodes)
	
	2.14

`$all/class[starts-with(@name, "de.tud.")]`
`  //invoke[@declaringClassName="de.tud.BytecodePointcut"`
`  and @methodName="getFilter"]`
 	
call(de.tud.BytecodePointcut.getFilter(..)) && within (de.tud..)*
	0.06
	(5 nodes)
	
	2.32

`let $types :=`
`  subtypes($all/class[@name="de.tud.Instruction"])/@name`
`return //invoke[@methodName="toString" and`
`  @declaringClassName = $types]`
 	
call(de.tud.Instruction+.toString(..))
	0.66
	(26 nodes)
	
	4.78

`pcflow(//method[@name="toString" and not(.//parameter[1])])`
`  //put`
 	
set(.*) withincode <pcflow(*.toString())>*
	10.64
	(14 nodes)

In order to show that our approach can be implemented with reasonable performance we made an initial performance evaluation of pointcuts as queries. We have measured the time needed to evaluate some pointcut queries using the BAT toolkit itself as the code base [8], which has 704 class files (roughly 1.5MB of uncompressed class files). The following table shows the evaluated queries along with an equivalent AspectJ PCD (if possible) and the time required to evaluate them together with the number of selected nodes. To measure the times for AspectJ we have used an instrumented variant of the weaving class loader in order to isolate the time to evaluate a pointcut as much as possible. We simply used a `declare warning` statement such that no actual weaving took place[9]. However, this should not be considered a hard, fair comparison because the time to load the XML representation takes longer than reading the binary representation of a class and more memory is required (roughly 3-4 times) to hold the XML representation. Further, the time measured for the AspectJ weaver includes the time to create its internal representation out of an in-memory byte array, whereas in our case the input is parsed XML which does not need to be further transformed. The purpose of these numbers is merely to indicate that the performance does not explode in our system for queries that resemble AspectJ's pointcut designators. Indeed, they indicate that the performance is reasonable and that the architecture can be used to prototype new queries and will deliver

[8] Time measured on an AthlonXP 2600, 512MB Ram, WindowsXP and Sun JDK 1.5.0beta1

[9] We used AspectJ version 1.2 for comparison.

comparable numbers. Note that for queries, such as e.g., the `pcflow` query, it is necessary to further investigate if they can be implemented efficiently enough for every day usage.

These preliminary measurements are very encouraging and show that the usage of a full-fledged query pointcut language can be considered.

4 Related Work

The most relevant related works are approaches that propose crosscut languages based on logic query languages [9, 6]. In these approaches, pointcuts are specified as logic queries in Prolog or Prolog-like languages. JQuery [9] is a browser which allows users to select views of the program and to define how the selected program elements should be ordered on the browser. The logic query pointcut language proposed in [6] operates on top of a reification of static and/or dynamic properties of Smalltalk programs.

The goal of these languages is quite similar to ours, the main difference being the usage of a logic versus a functional query language. The pointcut language of [6] can directly express dynamic join points, while the pointcut language discussed in this paper can only express patterns relating static join points. However, we think that this difference is mainly due to the different data model on which the languages operate.

A more important difference is that, by the usage of unification, one gets a powerful mechanism to retrieve context information from join points for free. This is more complicated in our approach, where context information has to be explicitly retrieved. On the other hand, in our approach one gets the advantage of a statically-typed functional language with a powerful module system for free. The possibility of creating user-defined pointcut libraries as in our approach is not mentioned in [6]. We cannot yet give a final answer to the question which of these approaches is better suited to express pointcuts, but we think that at least our work provides a good basis for a comparison.

Josh [2] is an AspectJ-like language with an extensible pointcut mechanism built on top of Javassist [3]. The Josh compiler takes an aspect as input and produces a special weaver – a Java program that uses Javassist to perform the weaving. The generated weaver iterates over all elements of a program that are exposed as meta-objects by Javassist, such as classes, fields, constructors and method objects, and adds the advice code if the element at hand represents a join point shadow [11] that matches the pointcut designator of the advice. To realize matching, the compiler generates calls to static methods corresponding to pointcut designators provided by Josh. Such methods are implemented by a developer in the process of extending Josh with new PCDs[10].

There are significant differences between Josh and our approach. Especially, Josh does not support declarative pointcut specifications. New PCDs are im-

[10] In addition to pointcut matching, the methods also take care of injecting dynamic checks as well as code needed for exposing context information at a join point to the advice code.

plemented as meta-programs in Josh using the Javassist library. Josh basically suffers from the problems of a meta-programming approach, especially with respect to the composability of the PCDs implemented as meta-programs. In the introduction, we claimed the precise specification of pointcuts as one of the benefits of the pointcuts-as-queries approach as compared to current AOP languages, where pointcuts have complicated, imperative semantics. This is worse in Josh, where such complicated imperative meta-programming semantics can be written by the developer.

Masuhara and Kawauchi presented in [16] a pointcut which selects join points based on the flow of data. To be precise, the proposed `dflow` pointcut designator selects join points with a specific value, if the value originated from a location selected by a pointcut. We think that our query language would be a good basis for specifying these kinds of pointcuts in a general-purpose framework, so this is part of our future work.

We view our approach as contributing to the definition of an AOP language capable to accommodate different join point models of different AOP approaches. In this respect, it is related to the *Concern Manipulation Environment* [8] (CME) which is intended to be an extensible, reusable, open, and customizable platform on which AOSD tool developers can build and integrate tools. Opposed to CME's goal to provide a query engine as one of the components (tools) of a concern manipulation environment, we argue for having crosscuts be first-class constructs of an aspect-oriented language and for having built-in language expressions that operate on these values. However, we also investigated the usage of queries over the program in an IDE in the context of the XIRC tool [4], which can be used to visualize crosscutting structure specified as a functional query.

We believe that an aspect-oriented language with a pointcut language such as our query language is general enough to express a wide range of very different pointcut models. Hyper/J [18] and Demeter [14, 17, 12] are two approaches to AOP that have a very different pointcut model than AspectJ. Since Hyper/J uses only static composition rules, we think that these rules can be directly expressed as queries. For example, a composition rule such as *mergeByName* boils down to finding methods/classes with the same name, which can easily be expressed as a query.

Demeter is related in that both approaches use a notion of path expressions, called *traversal strategies* in the context of Demeter. The aim is different, however. In Demeter, path expressions are used to find traversal paths in the object graph whose edges are association and inheritance links, while we use path expressions over the program tree for selecting arbitrary sets of related join points. Nevertheless, we think that it is possible to use a query to find traversal paths in the sense of Demeter. We plan to create a query library for traversal strategies in the future. In [13], Lieberherr tries to establish a common framework for relating pointcut expressions a la AspectJ and traversal strategies as they are used in Demeter, stating that in his view, both can be described by a two level graph structure, with a selection language as the top-level. The work presented here makes this relation very explicit.

5 Summary

In this paper, we have investigated the usage of the functional query language XQuery for the specification of pointcuts. We have shown that XQuery enables powerful composition and reusability mechanisms for pointcuts. While we have investigated this approach only in terms of a static representation of the program, we think that the general idea can be generalized to arbitrary representations of the program semantics.

The aim and achievements of our work are similar to other approaches using logic query languages. To a degree, a comparison between these approaches boils down to a comparison between the respective paradigms, logic programming versus functional programming. We hope that our approach will be a good basis to conduct this debate in terms of the specific requirements of a pointcut language.

References

1. Christoph Bockisch and Michael Eichberg. BAT. http://www.st.informatik.tu-darmstadt.de/bat, 2004.
2. Shigeru Chiba and Kiyoshi Nakagawa. Josh: An Open AspectJ-like Language. In *Proceedings of AOSD 2004*, Lancaster, England. ACM Press.
3. Shigeru Chiba and Muga 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.
4. M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. XIRC: A kernel for cross-artifact information engineering in software development environments. In *Proceedings of 11th IEEE Working Conference on Reverse Engineering (to appear)*, 2004.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, 1995.
6. Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern–Based Crosscuts. In *Proceedings of AOSD 2003*, pages 60–69, Boston, Massachusetts. ACM Press.
7. Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In *Proc. of AOSD 2004*. ACM Press.
8. IBM Watson Research Center. Concern manipulation environment (CME): A flexible, extensible, interoperable environment for AOSD. http://www.eclipse.org/cme/.
9. Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187. ACM Press, 2003.
10. Gregor Kiczales. Keynote talk at AOSD 2003. http://www.cs.ubc.ca/∼gregor/.
11. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffry Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, Budapest,Hungary. Springer.
12. Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, 44(10):39–41, 2001.

13. Karl J. Lieberherr. Controlling the Complexity of Software Designs. Keynote talk at ICSE'04.
14. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.
15. Hidehiko Mashuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In *Foundations of Aspect-Oriented Languages Workshop at AOSD '02*, 2002.
16. Hidehiko Masuhara and Kazunori Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *Proceedings of APLAS'03*, Lecture Notes in Computer Science, pages 105–121, Bejing, China.
17. Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag. 8 pages.
18. Harold Ossher and Peri Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proc. of ICSE '00*. ACM Press, 2000.
19. World Wide Web Consortium. XQuery 1.0: An XML query language, W3C working draft Jun 7, 2001, http://www.w3.org/tr/xquery/, 2001.
20. World Wide Web Consortium. XQuery 1.0 formal semantics, http://www.w3.org/tr/query-semantics, 2001.