# Aspect-Oriented Programming: Selecting and Exposing Object Paths

Mohammed Al-Mansari, Stefan Hanenberg, Rainer Unland

University of Duisburg-Essen, Schützenbahn 70,
45117 Essen, Germany
{Mohammed.Al-Mansari,Stefan.Hanenberg,Rainer.Unland}
@icb.uni-due.de

**Abstract.** Aspects require access to the join point context in order to select and adapt join points. For this purpose, current aspect-oriented systems offer a large number of pointcut constructs that provide access to join point information that is *local* to the join point context, like parameters in method call join points. However, these systems are quite miserly with *non-local* information that cannot directly be derived from the local execution context. Recently, there have been some proposals that offer access to some kind of non-local information. One such proposal is the *path expression pointcut* that permits to abstract over *non-local object information*. Path pointcuts expose non-local objects that are specified in corresponding path expression patterns. In this paper, we show recurrent situations where developers need to access the whole *object paths*, and consequently, they add workarounds other than pointcut constructs to get the required accesses. Then, we present and study an extension to the path expression pointcuts to permit exposing the object paths and show how this extension overcomes the problem.

## 1 Introduction

Aspect-oriented programming aims to increase the modularity of software. This is achieved by features that are used to select points in the execution of the program and to adapt them. Pointcuts are the language constructs used for selecting these points, which are called *join points* [15]. The join point adaptation is achieved by the so-called advice. The selection and the adaptation of a join point depend on the characteristics of this join point.

These characteristics are called *join point properties* [1, 12] and are divided into two categories: Local and non-local join point properties depending on whether they are accessible and derivable from the join point context or not, respectively. For example, the current executing object at a method execution join point is considered a local join point property, which can be accessed with the `this` pointcut in AspectJ [15]. In general, current aspect-oriented systems offer several pointcut constructs that can be used to derive the local information of a join point. On the other hand, these systems provide only a small number of pointcut constructs that provide access to the non-local join points properties, like the call stack in languages such as Java and C++.

One intention of aspect-oriented systems is to provide pointcut languages that permit the developer to specify expressive pointcuts [27] where the join point selections correspond to the developer's mental model [32]. This also implies that the available join point properties provided by an aspect-oriented system must suffice the developers' needs. In addition to that, the resulting aspects would be easier to maintain, more robust against changes, and contain no mixture between the pointcut and advice code. As a consequence, it has been pointed out by a number of researchers that there is a need for more pointcuts that offer abstractions over non-local properties [2, 23, 35]. However, none of these proposals provide abstractions over the non-local properties that are based on object information, for which we proposed the path expression pointcut [1] as an explicit construct.

In this paper we take a step forward by extending the path expression pointcut that allows aspects to access the whole object information in the matching paths. We motivate our proposal using two examples from the object persistence concern. These examples illustrate why we need to get not only object references from the whole path but also field information that establishes the relationships between these objects. This is achieved by exposing the whole non-local part of the object graph that is related to the join point to the aspect.

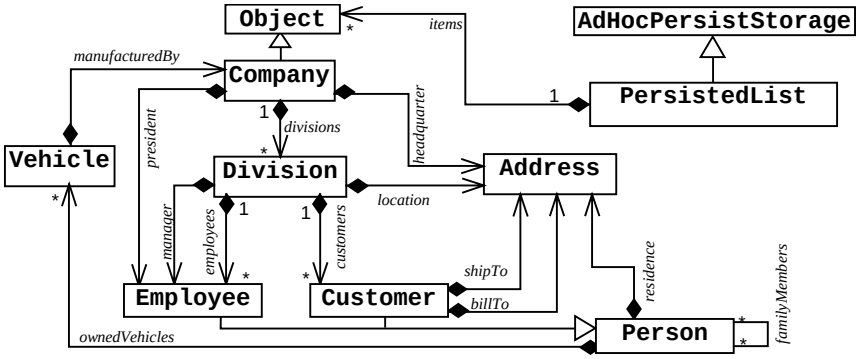



**Fig. 1.** Class diagram for the problem domain

The rest of the paper is organized as follows: In the rest of this section, we first describe the object model of an example that is used throughout the paper. Then for the purpose of self-containedness, we give a brief description on the current version of the path expression pointcut. In section 2, we discuss two motivating examples and the problem statement. Section 3 presents the extension to the path expression pointcut. We talk about related works in section 4. In section 5 we discuss some issues regarding our proposal and its implementation. The paper is concluded in section 6.

### 1.1 Example

Figure **1** shows a class hierarchy adopted from [17] for a company object model. The intention is to persist all company objects that are added to special containers called "persistent lists": Instances of `PersistedList` class that persist all contained objects.

These objects are not prepared to be persistent; rather they become persistent at the time they are added to the lists. This persistence service is an ad-hoc functionality provided by the persistent lists in a similar way to *spontaneous containers* [29].

As Figure **1** elaborates, each company has a number of `divisions`, a `president` and a `headquarter` address. Each division in turn has a `manager`, a number of `employees`, a number of `customers` and a `location` address. `Customer` (which we added) and `Employee` are subtypes of `Person`. Each person object is associated with a collection of `Vehicle` objects. Finally, vehicle objects are associated with corresponding company objects by the relation `manufacturedBy`.

### 1.2 Path Expression Pointcut

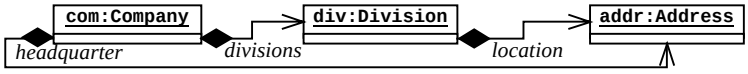
The path pointcut traverses the current object graph in order to find paths that match a given path expression. A pointcut making use of a path pointcut picks out the join points where there exists at least one matching path. The general form is:

```
path(PathExpressionPattern);
```

For a detailed description of the syntax, please refer to section 3.1 in [1]. Within path expression patterns may specify certain objects as source objects, target objects and intermediate objects of the paths. Moreover, the associations between objects can be specified by names. The path pointcut applies pattern matching mechanism by using the wildcards "`*`" and "`/`" to specify associations between objects along the path. The path pointcut is used to expose both local and non-local context from the join points and it can be used along with other pointcut designators by means of operators "`&&`", "`||`" and "`!`". For example, consider the following pointcut:

```
pointcut pc(Company c, Person p, Object o):
  path(c -*-> Employee p -/-> o) && set(* *) && target(o);
```

This pointcut picks out every set join point whose target is the object `o` and where there is at least one path between the objects `c` and `o` via `p`. The wildcard "`*`" in the path expression indicates that there is a direct relationship between `c` and `p`, whereas "`/`" indicates that there may be many objects on the path between these objects.



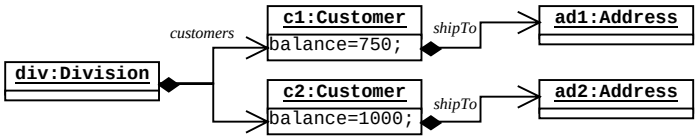

```
pointcut pc(Company c, Address a):
  path(c -/-> a) && target(a) …;
```

**Fig. 2.** A `Company` instance has two references to the same `Address` instance

Like in AspectJ, the objects described in the path expression can be bound to the corresponding variables in the pointcut's header. Then the bounded objects are exposed from the join point context to the aspect context. According to the semantics of the path pointcut, the result of a path expression is a set of distinct valid parameter bindings rather than a set of matched paths.

For example, in Figure **2**, the path expression in the pointcut matches two paths: `(com –headquarter-> addr)` and `(com –divisions-> div -location-> addr)`, however, the valid parameter bindings is only: `(c=com, a=addr)`. Notice that there are two occurrences of the variable name `a`, once as a destination in the path expression and then it is used in the `target` pointcut. The path pointcut allows multiple occurrences of the same variable name in one or more path expressions and unifies these occurrences to be bound to the same value.

A consequence of multiple parameter bindings is that the advice execution mechanism is modified to allow a single advice that is associated with the pointcut to be executed as many times as the size of the parameter bindings set.




```
pointcut pc(Division d, Customer c, Address a):
  path(d -/-> c -/-> a) …;
```

**Fig. 3.** Two resulting paths between a division and an address are resolved to two bindings

For example, according to the pointcut of Figure **3**, the resulting bindings are: `(d=div, c=c1, a=ad1)` and `(d=div, c=c2, a=ad2)`. Hence, each single advice that is associated with this pointcut must be executed two times for each single binding. An important question is in which order these executions should be performed. For example, assume that there are two concurrent transactions, each updates one of the objects `ad1` and `ad2` and that the developer wants to run these concurrent changes in a descending order by customers balances. Since the balance of `c2` is greater than that of `c1`, the advice must be executed first with the binding: `(d=div, c=c2, a=ad2)`.

```
public boolean addrChg(Object[] o1, Object[] o2) {
  Customer cust1 = (Customer) o1[1];
  Customer cust2 = (Customer) o2[1];
  return cust1.getBalance() > cust2.getBalance();
}
pointcut pc(Div d, Customer c, Address a):
  set(* *)&& target(a)&& path(d -/-> c -/-> a) orderBy(this.addrChg);
```

**Fig. 4.** Possible ordering method specification

As a solution, we provide an extra construct, namely `orderBy` that is added to the path pointcut. It takes a name of the method containing the ordering code specified by the developer. The method is similar to the `compare` method of the `Comparable` interface in the Java API. For example, in the pointcut `pc` of Figure **4**, the parameter of the `orderBy` is the name of the method `addrChg` whose signature has two array parameters of type `Object` each representing a single binding. The method extracts the second element from both arrays, casts them to type `Customer` and returns the comparison result between their `balance` fields. When no `orderBy` clause is specified, then the order would be undefined.

## 2 Motivation

One of the important issues of object persistence is to ensure the isolation property of concurrently executing transactions by means of concurrency control approaches. The concept of locking data items is one of the main used techniques of the concurrency control. There are two main types of locks: Shared or exclusive, generally known as read and write lock respectively. In what follows, we will consider two locking policies in our motivating examples: The field-based locking policy and the cascading-version locking policy.

### 2.1 Example 1: Field-Based Locking

In the locking-based concurrency control of transactions literature, there is a large number of researchers discussing locking granularities [11], proposing techniques for fine-granularity locking [24, 28] and discussing the benefits and the effects of multiple locking granularities [30]. The granule of the data that can be locked is either the whole database, an extension of objects, an object or a field of an object [14]. Here, we focus on locking the fields of the object that are being changed so that multiple transactions can be executed on this object.

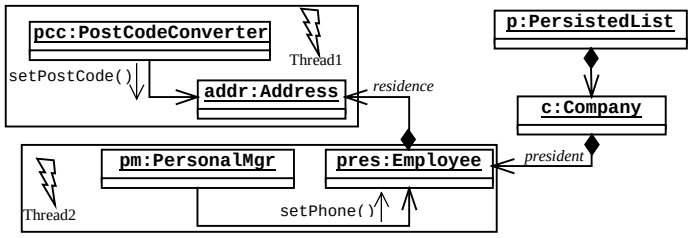



**Fig. 5.** Two separate concurrent transactions attempt to change the `pres` state

In Figure **5**, the `Company` instance `c`, which references the object `pres`, is added to the persistent list `p`. Hence, this company object and all objects in its closure are to be made persistent. Assume that we are interested in applying locking on the fields of the `Employee` object. Two separate concurrent threads attempt to update the state of the employee. Thread1 in `pcc` wants to change the postcode value of `addr` and Thread2 in `pm` attempts to change the `phone` field of the employee. In order to permit both threads to modify the employee object simultaneously, we should acquire separate write locks for the fields `residence` and `phone` rather than for the object `pres`.

In aspect-oriented terms, each update is a join point that should be selected since the employee object belongs to the persistent list `p`. The needed information is: the objects `p` and `pres`. The aspect then should acquire a write-lock for each field that is to be changed. Here, the needed information is the fields `residence` and `phone`.

The local relevant context at the join point in Thread1 is the object `addr`, and the local relevant context for Thread2 is the object `pres`. In Thread1, the objects `p` and

pres, in addition to the field name residence, are considered to be non-local in Thread1, while the non-local object to Thread2 is p.

```
pointcut pc1(PersistedList pl, Employee e):
   set(* *) && target(e) && path(pl -/-> e);

pointcut pc2(PersistedList pl, Employee e, Object o):
   set(* *) && target(o) && path(pl -/-> e -/-> o);

before(PersistedList pl, Employee e): pc1(pl, e) {
  String fname = thisJoinPointStaticPart.getSignature().getName();
  Field field = … // code to get the field by using its name and lock it
}
before(PersistedList pl, Employee e, Object o):
  pc2(pl, e, o) {
  // code for getting the field of e which is the beginning of the reference closure from e to o
}
```

**Fig. 6.** Using the path pointcut in the field locking example

Consider the pointcut pc1 in Figure **6**, it selects all set join points where the target object e of type Employee is reachable from the persistent list pl. According to Figure **5**, pc1 selects the employee object pres due to the matching path: (p -> c -> pres), and the resulting bindings is: (pl=p, e=pres), which is exposed to the before advice. The advice gets the changed field by means of the reflective facilities in AspectJ and Java. Finally, the advice acquires the write-lock to the field phone of the object pres.

The pointcut pc2 selects all set operations targeted to any object o that is reachable from the persistent list pl via object e. From Figure **5**, there is one matching path to the path expression pattern in pc2, i.e. (p -> c -> pres -> addr). Hence, this resulting binding is (pl=p, e=pres, o=addr), which provides the advice with the access to the non-local objects p and e in addition to the local object addr. The advice must acquire a write-lock for the dirty field residence, however, this field is not available for the advice since this information is non-local to the join point.

To get access to the non-local field information, developers implemnt workarounds since this information cannot be accessed by using the available pointcut construct. These solutions are complex, difficult to maintain and error-prone though. Moreover, their code does not reflect on the semantics of the join point selection and adaptation.

### 2.3 Example 2: Cascading Version Locking

To solve the concurrency control problem, researchers also proposed a number of version-based locking policies [18, 22, 25]. In these policies, all transactions can grant shared read access to the object, and whenever a transaction attempts to update the state of the object, the application should check whether this update is performed on the right version of the object.

Version locking mechanisms use a so-called version (or write-lock) field that is added to every object and compare this field every time an update operation on the object is committed with the current value in the datastore. If they are equal, then the change is committed to the datastore, otherwise the change is disallowed and this indicates that the object must have been updated by another transaction. In the cas-

cading version locking, the version field of all objects that reference the dirty object must be updated also.

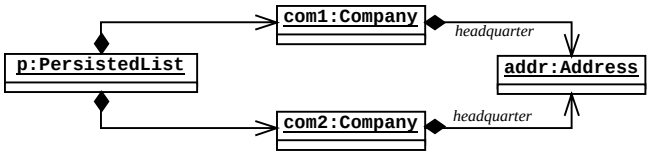



**Fig. 7.** A shared `Address` instance between two `Company` instances

As an example, consider the object graph in Figure 7. Two `Company` instances, `com1` and `com2`, both referencing the same object `addr`. The `Company` instances are stored in the persistent list `p`. According to the version locking policy, any change to the `addr` will update the version field of `addr` as well as the version fields of its owner objects, i.e., `com1` end `com2`. In our aspect, we want to be sure that the changed object is reachable from a persistent list. If so, the aspect should perform the dedicated version locking policy on the shared object. So, we use the following pointcut:

```
pointcut pc(PersistedList pl, Object o):
  set(* *) && target(o) && path(pl -/-> o);
```

A corresponding advice gets access to the binding `(pl=p, o=addr)`, despite the fact that there are two different paths from `p` to `addr`. The advice will consider the changes in the `Address` object and update its version field. However, the corresponding version field of `com1` and `com2` are not modified yet. Such situations affect the data consistency. In order to overcome this problem, we must get access to all objects in each path from `p` to `addr` in order to modify their versions.

As mentioned in the first example, the only way currently available for the developer in conventional aspect-oriented systems is to apply introspective facilities of the language to traverse the entire reference path to get the required accesses. These kind of solutions are not trivial, error-prone and mostly not reusable.

In summary, both examples illustrate the need for more expressive path pointcuts. The first example motivates the need for exposing not only the non-local objects, but also the non-local field information. The second example motivates the need for exposing all objects in the matching paths instead of the distinct parameter bindings.

## 3 Extended Path Expression Pointcuts

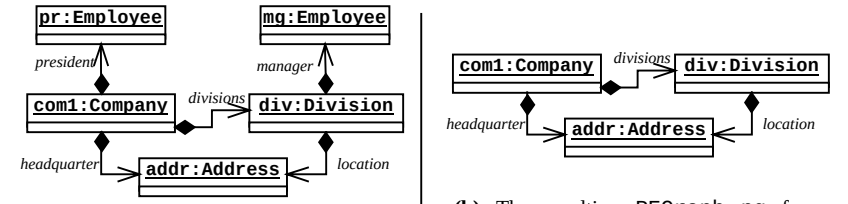
In this section, we present an extension to the path expression pointcuts that overcomes the problems described above. We modified the syntax and the semantics of the pointcut construct so that the resulting paths are exposed to pointcuts and advice as a subgraph of the whole object graph. This subgraph is made local to the aspect and from its interface developers can extract the objects and their relationships.

### 3.1 Syntax and Semantics of the Path Pointcut

In order to get access to the resulting paths, we slightly modified the syntax of the path expression pointcut so that it has two parameters: The first parameter refers to an instance of type `PEGraph` (discussed in section 3.2). The second parameter is the path expression pattern. The new syntax is:

```
path(PEGraph identifier, PathExpressionPattern).
```

The new model maintains the syntax specifications of the `PathExpressionPattern` discussed in [1]. The path pointcut calculates the path expression pattern against the current heap and adds all matching paths to a generated `PEGraph` object. When the evaluation process ends, the resulting `PEGraph` object is bound to the variable name `identifier`. The pointcut header must include this variable name as a parameter of type `PEGraph`. This parameter will be added to resulting parameter bindings being exposed to the pointcut and the associated advice.




**(a)** The object graph

**(b)** The resulting `PEGraph pg` from pointcut `pc`

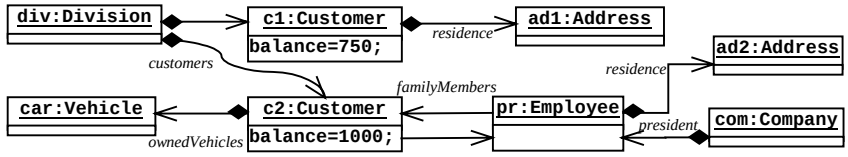
```
pointcut pc(PEGraph pg, Company c, Address a):
  path(pg, c -/-> a) …;
```

**(c)** The resulting bindings: `pg, c=com, a=addr`

**Fig. 8.**Two paths between a `Company` and an `Address`

As an example, consider the object graph in part (a) of Figure **8**. There are two matching paths to the given expression in the path pointcut: `(com -divisions-> div -location-> addr)` and `(com -headquarter-> addr)`. The evaluation of this path expression creates an object of `PEGraph` that consists of the subgraph in Figure **8**-(b). This object is bound to the variable `pg`. Finally, as shown in part c), the pointcut `pc` resolves the parameter bindings `(pg, c=com, a=addr)`.

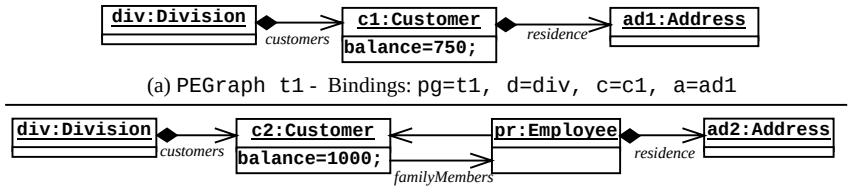
The created `PEGraph` at a given join point depends also on the resolved bindings. That means each distinct parameter binding has its own corresponding `PEGraph` object, which ensures exposing only relevant information to the join point. The relevant information consists of the objects and their relationships that are included in the matching paths even if these paths contain cycles. Notice that in Figure **8**-(b), the two `Employee` objects along with their referencing field information, i.e. `president` and `manager`, are excluded from the result of the path pointcut since this information is not relevant to the given path expression.




```
pointcut pc(PEGraph pg, Division d, Customer c, Address a):
  path(pg, d -/-> c -/-> a) …;
```

**Fig. 9.** Infinite number of paths between objects `div` and `ad2`

The presence of cycles in object graphs raises an important question regarding the termination of our pointcut construct. If a cycle appears in a matching path then it must be included in the resulting path graph. To guarantee the termination feature of the path pointcut, cycles should not be traversed more than once except when it is needed to traverse them again to fulfill the required set of bindings. Notice that we have to detect the cycles during the traversal of the object graph and have to add them to the resulting `PEGraph` if they occur in a matching path.




(a) `PEGraph t1` - Bindings: `pg=t1, d=div, c=c1, a=ad1`

(b) `PEGraph t2` - Bindings: `pg=t2, d=div, c=c2, a=ad2`

**Fig. 10.** Two `PEGraph` objects, each for a different bindings

Consider the collaboration diagram in Figure **9**. According to the given path expression, there is one matching path from `div` to `ad1` via `c1`, which constructs a temporary `PEGraph` object `t1` that is bound to the variable `pg`. The resolved binding is: `(pg=t1, d=div, c=c1, a=ad1)` as shown in Figure **10**–(a). On the other hand, there is an infinite number of matching paths from `div` to `ad2` via `c2` due to the presence of the cycle between the objects `pr` and `c2`. Suppose that the traversal algorithm visits `div` then `c2` and finally reaches `pr`, if it selects to go through the edge labeled `familyMembers` then it will visit `c2` again, detect the cycle, save all information (objects and relations) and finally will return back to `pr` and follow the other edge to the object `ad2`. At this point the traversal algorithm finds a matching path as well as a valid distinct bindings for `d`, `c` and `a`. The whole path along with the cycle will be put in a temporary `PEGraph` object `t2` as in Figure **10**–(b), and then the pointcut resolves the second binding: `(pg=t2, d=div, c=c2, a=ad2)`. The pointcut evaluation stops afterwards since there are no more matching paths.

Last but not least, the new extension maintains the semantics of the advice execution in the first version of the path pointcut. Since the number of distinct resulting bindings is two in the last example, any advice associates with the pointcut `pc` executes two times. The ordering schema discussed in section 1 also applies here. Hence,

if the developer wants to run the advice first on the bindings that contains higher balance customers, then (s)he must define the same ordering method of Figure **4** except that the index of the customer object in the binding is 2 instead of 1.

### 3.2 Path Expression Graph

As stated above, the result from the path pointcut is a subgraph of the whole object graph that contains only the information relevant to the selected join point. We said that this resulting graph will be assigned to an object of type `PEGraph`. The public interface of this data structure is illustrated in Figure **11**.

```
class PEGraph<Current, Next> {
  private Current current;
  private Next next;
  public Current getCurrent();
  public Next getNext();
  public List<PENode> getNextNodes();
  public List<PEEdge> getNextEdges();
  public boolean setCurrentObject(Current current);
}
class PENode<T> {
  public T getObject();
  public List<PEEdge> getOutEdges();
  public List<PEEdge> getInEdges();
}
class PEEdge<T> {
  public PENode<T> getRelatedNode();
  public String getRelField();
}
```

**Fig. 11**. The public interface of the `PEGraph`

Each node of the `PEGraph` is of type `PENode` and contains a reference to an object of a generic type, a list of outgoing edges and a list of incoming edges. The edge of the `PEGraph` is of type `PEEdge`, which has two methods: `getRelatedNode` to get the generic typed object that represents the related node to the owner node of this edge (i.e. the node at the other edge). The method `getRelField` returns the relation name. The graph object provides the ability to navigate from any given object either in a forward or in a backward manner. This is done by getting access to the current object of the `PEGraph` instance, and then accessing its related objects and referencing fields.

The `PEGraph` interface cannot be mutated other than setting the current object field of the `PEGraph` class by using the method `setCurrentObject`. It is possible to set the current node by passing a reference to a specific object or a given `PENode`. The methods `getNextNodes`, `getNextEdges` and `getNext` are used to traverse through the `PEGraph`. It should be mentioned that the object and field information returned from these methods is obtained from the resulting path expression graph. Other object information from the whole object graph that is not related to the selected join point could be accessed from the object that is associated with the `PENode`. The method `getCurrent` of the `PEGraph` returns the current object as a `PENode`, which can be used by the developer to get the object being associated with this node directly with help of method `getObject`.

This representation is making use of generic types, which allow developers to use the type information they know, either directly or from the `PEGraph` object, without casting. For example, consider the following pointcut specification:

```
pointcut pc(PEGraph<PENode<Division>, PEGraph<PENode<Customer>>> pg,
            Division d, Customer c): path(pg, d -*-> c) …;
```

Here, the source of any matching path is specified to be of type `PENode<Division>`. Hence the following is type-safe:

```
Division myDiv = pg.getCurrent().getObject();
```

We try to make it as easy as possible for the developer to query over the dynamic object information that is relevant to the join point. Of course, there is a complexity overhead in our representation of the `PEGraph`, however, this is significant from the developer's point of view since it needs less effort to reason about type information (as compared to performing reflective operations and casting operations).

```
pointcut pc1(
   PEGraph<PENode<PersistedList>, PEGraph<Object, Object>> pg,
   PersistedList pl, Employee e):
  set(* *) && target(e) && path(pg, pl -/-> e);
pointcut pc2(
   PEGraph<PENode<PersistedList>, PEGraph<Object, Object>> pg,
   PersistedList pl, Employee e, Object o):
  set(* *) && target(o) && path(pg, pl -/-> e -/-> o);
before(
   PEGraph<PENode<PersistedList>, PEGraph<Object, Object>> pg,
   PersistedList pl, Employee e, Object o):
 pc2(pg, pl, e, o) {
 pg.setCurrentObject(e);
 List<String> dirtyFields = pg.getNextFields();
 for(String dfn: dirtyFields) {
   // get the field from the object e and  acquire a write-lock for it …
 }
}
```

**Fig. 12.** The `PEGraph` object is used in the path pointcut in `pc1` and `pc2`

### 3.3 Field-Based Locking Example Revisited

We use the new path pointcut in the pointcuts `pc1` and `pc2 as` shown in Figure **12**. According to the object graph in Figure **5**, the path expression in pointcut `pc1` matches the path: `(p -items-> c -president-> pres)`. The `PEGraph` object `pg` will contain this path and it will be exposed to the advice along with the bindings `(pl=p, e=pres)`. The associated advice would adapt the join point as shown in the motivating example since the `pres` object and the field `phone` is local information to the join point. It must be noted that the aspect maintains the `PersistedList` objects from which the traversing process begins.

When applying the pointcut pc2 to the collaboration diagram in Figure **5**, the only matching path is: `(p -items-> c -president-> pres -residence-> addr)` that

represents the resulting `PEGraph` object `pg`. This object along with the objects `p`, `pres` and `addr` are exposed to the associated before advice in the figure. The first line of the advice sets the current object of the `pg` to `e`, which is the variable bound to the `Employee` object `pres`. The second line gets all fields of the current object `pres` that are available in the `pg` graph and puts the result in the array `dirtyFields`. The rest of the advice is the code responsible for iterating through the list elements and acquiring a write-lock for each. The only available field according to the resulting path graph in `pg` is the field `residence` of the object `pres`.

### 3.4 Version-Based Locking Example Revisited

As in the last section, the extended path pointcut provides us with access to the required non-local information in the example: First, the persistent list object that contains the changed object, second, all objects in all reference paths from the persistent list to its contained changed object.

The following pointcut is making use of the path pointcut construct:

```
pointcut pc(PEGraph<PENode<PersistedList>, PEGraph<Object, Object>>
pg, PersistedList pl, Object o):
  set(* *) && target(o) && path(pg, pl -/-> o);
```

With respect to Figure **7**, the resulting paths from this pointcut are: `(p -items-> com1 -headquarter-> addr)` and `(p -items-> com2 -headquarter-> addr)`. These paths construct the part of the object graph that will be exposed to the advice as illustrated in Figure **13**. This subgraph contains the information relevant to the set join point on the address object, that is, all objects that are referencing the changed object `addr`. The pointcut `pc` resolves one parameter binding: `(pl=p, o=addr)`, which is exposed to the advice, which will be executed just once.

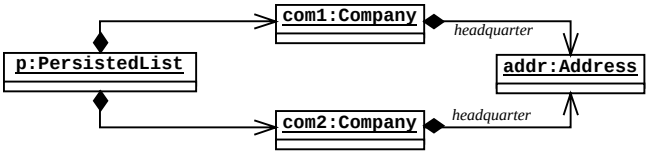



**Fig. 13.** The relevant `PEGraph` from the object graph in Figure **7**

Since the whole reference paths are exposed to the advice, performing the cascading version locking would be trivial. The only thing developers have to do inside the advice is backward traversing the nodes of the `pg` object after setting the current object of the `pg` to point to `addr`. They can easily update the version fields of all traversed objects in order to apply the cascading version locking. Note that although the paths in this example are of length 2, this algorithm is useful for any length of the reference path where there is a guarantee to get access to all owner objects of the current object.

## 4 Related Work

Path expressions, first introduced in [5] to synchronize the operations on data objects, then became central ingredient of object-oriented query languages such as [10, 17, 34]. The W3 Consortium introduced the XPath language [6] in order to address parts of an XML document [4]. In our work, we study the benefit of the application of path expressions in increasing the expressiveness of pointcut languages in addressing object relationships at runtime and providing aspects with access to this information.

Adaptive programming (AP) [20, 21] and strategic programming (SP) [19] provide interesting notions similar to the path expressions. They provide the developer with traversal control by the help of so-called traversal strategies and traversal schemes, respectively. The idea behind the aspect versions of AP and SP is that the advice is executed whenever the visitor component visits an object that belongs to a path that matches the given traversal. This is in contrast to the path pointcut that participates in the selection of the join point and exposes the matching object paths as well.

A lot of research effort is done to provide access to the non-local join point properties. Some works cover the importance of selecting and adapting the join points based on execution trace matching. Stateful aspects [8, 35] define conditions based on finite state transitions to trigger advice executions on a protocol sequence of join points. Other trace-based solutions have been discussed in [2, 9, 36]. Data flow pointcuts [23] solve the problem of non-locality of data flow information at the join point. The context-aware aspects [33] provide means to access information that is associated with certain contexts that are currently available or occur in the past. Another well-known example of accessing the non-local call stack at the join point is the `cflow` pointcut in AspectJ. However, all these proposals neither point to nor solve the problem of non-local join point properties that are based on object information.

Some works already discussed the need for expressive join point models that reflect the mental model of the developer [32]. Moreover, expressive pointcuts increase the modularity [27] and are robust to absorb any changes to the application features and compositions. The authors in [27] followed their previous remark about pointcuts that access dynamic properties of the program [3] by implementing the Alpha language whose pointcuts are Prolog queries over a database consisting of different semantic models of the program execution. In our proposal we have to keep small parts of the current heap at a given join point. In contrast to logic-based pointcuts, the path pointcut relies on traversing the heap to get relevant object information in a form of paths. Alpha predicates can be used to compose pointcuts that represent the notion of path pointcuts, however, these compositions may result in complex pointcut definitions that can be avoided by using the path pointcut. Moreover, one of the main goals of our proposal is to apply the path expressions technique to AOP as an explicit construct and to discuss the effects of this integration and how to resolve them.

In association aspects [31], a new pointcut designator is introduced to AspectJ, namely `associate`, which is used to associate extended per-object aspect instances to a group of objects. The authors point to the need for multiple executions of the advice in the associated aspect instances. This multiple advice execution corresponds to our approach, however, they do not give a clear specification of the order in which these executions run such as what we have proposed in this paper.

## 5 Discussion

Objects graphs are directed and may contain cycles, which is a source of complexity since there is a infinite number of matching paths to a given path expression in such structure. We can minimize this complexity by considering some restrictions, e.g., finding paths in the object graph that contain a cover to the needed bindings. This can be achieved using any of the efficient algorithms to detect cycles in directed object graphs and stop traversing through these cycles. For example, the time complexity of Floyd's cycle-finding algorithm [7] is $O(V)$, where $V$ is the number of nodes in the graph. Moreover, this guarantees the termination of the path evaluation process.

Depending on the path expression, there are different situations of complexity. If the source and the target objects of the path expression are specified, then the problem is minimized to the single-pair shortest path, which is faster than the cases where one or both objects are not know. The running time is ranging from $O(V \lg V + E)$ to $O(V^2 \lg V + VE)$ or even to $O(V^3)$ in the worst case for finding all paths between any two objects [7], where $V$ and $E$ are the number of nodes and edges of the object graph respectively. In fact this complexity also depends on the algorithm being used and the type of the object graph. For example, in sparse graphs, where $E$ is much less than $V^2$, Johnson's algorithm runs faster than the Floyd-Warshall algorithm.

The here proposed extension for the path pointcut maintains the same ordering schema of the multiple advice executions at a given join point. I.e., the developers are asked to define their own ordering rules in a separate method. Following our argument about the rationale behind this design choice in [1], we still believe that it is not difficult to define reasonable ordering methods that ensure the termination.

One important issue when dealing with the object graphs in persistence systems is how these systems manipulate the collection objects. Most persistence systems consider these objects as second-class objects, e.g., [13]. When two objects are sharing a second class object then each will have its own copy of this shared object so the changes in one copy will not be observed by the other owner object. Consequently, there must be a clear definition of how to represent these objects in the resulting `PEGraph` from a path pointcut. For simplicity purposes, we ignore this issue in this paper and treat items of the collection individually.

## 6 Conclusion

In this paper, we continue our argumentation about the need for abstractions in aspect-oriented systems that provide access to the non-local join point context. The motivating examples show the need to access non-local join point context that is not only based on objects but also on the relationships between these objects. Our examples cover some recurrent situations that occur when applying well-known locking policies in object persistence systems in an aspect-oriented manner. We have considered two such policies, field-based locking and cascading version locking mechanisms. For the first policy, we illustrate the need for accessing the non-local field information to apply the desired locking. In the second case, we show the need for getting the whole resulting paths instead of getting access to some objects in the path.

Then we describe our solution to such problems as an extension for the path expression pointcut [1]. The new extension provides access to the whole part of the object graph that is related to the join point and that is constructed from the matching paths. This subgraph is exposed then by the path pointcut to the aspect. Then we discuss some issues related to the extension. These include the public interface of the constructed subgraph that the developers can use to reason about objects and their relationships inside their aspects. We give our arguments about this representation. Then we discuss various aspect-oriented programming concepts and how the path pointcuts influence them with the help of some illustrative examples.

In the last section we present a general discussion about our proposal and some of its weaknesses. We suggest some possible ideas how to minimize the effects of these problems. These points are the focus of our future work.

Our experience with the path expression pointcuts shows a reasonable number of cases that require access to non-local object relationship information in persistence systems. However, we expect that path pointcuts have a large impact on increasing the expressiveness of pointcut languages. The reason for this is the flexibility of the path pointcut to express the mental model of the developers upon the role of object relationships in join point selection.

## References


1. Al-Mansari, M., Hanenberg, S.: Path Expression Pointcuts: Abstracting over Non-Local Object Relationships in Aspect-Oriented Languages; NODe'06, Erfurt, Germany; Sept 2006, LNI vol. P-88, pp. 81-96.
2. Allan, C., Augustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhot´ak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In OOPSLA '05, San Diego, California, October 2005, ACM Press, pp. 345-364.
3. Bockisch, C., Mezini, M., Ostermann, K.: Quantifying over dynamic properties of program execution. In 2nd Dynamic Aspects Workshop, Chicago, Illinois, March 2005, pp. 71–75.
4. Bray, T., Paoli, J., Sperberg-McQueen. (eds.): Extensible Markup Language. Available online at http://www.w3.org/TR/REC-XML, 1998.
5. Campbell, R., Habermann, A.: The Specification of Process Synchronization by Path Expressions. LNCS (Eds. G. Goos and J. IIartmanls), pp. 89-102, V16, Springer Verlag, 1974.
6. Clark, J., Derose, S. (eds.):. XML Path Language (XPath), version 1.0. Available online at http://www.w3.org/TR/Xpath, 1999.
7. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill, 2001.
8. Douence, R., Fradet, P., Südholt, M.: Composition, Reuse and Interaction Analysis of Stateful Aspects. In Proceedings. of AOSD'04, Lancaster, UK, March 2004, pp. 141-150.
9. Douence, R., Fradet, P., Südholt, M.: Trace-based aspects. In Filman, R. E., Erlad, T., Clarke, S., Aksit, M.. (eds.): Aspect-oriented Software Development, Addison-Wesley, 2005, pp. 201–217.
10. Frohn, J., Lausen, G., Uphoff, H.: Access to Objects by Path Expressions and Rules. I Proc. Of VLDB'94, Santiago, Chile, September 1994, pp. 273-284.
11. Gray, J., Lorie, R., Putzolu, G., Traiger, I.: Granularity of Locks in a Large Shared Data Base. VLDB 1975, pp. 428-451.
12. Hanenberg, S.: Design Dimensions of Aspect-Oriented Systems. PhD dissertation. Duisburg-Essen University, October 2005.

13. Jordan, D., Russell, C.: Java Data Objects, O'Reilly Media, 1st edition, 2003.
14. Kemper, A., Moerkotte, G.: Object-Oriented Database Management: Applications in Engineering and Computer Science. Prentice Hall, 1994.
15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G.: An overview of AspectJ. In ECOOP '01, Budapest, Hungary, June 2001, LNCS 2072, pp. 327-353.
16. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In ECOOP'97, Jyväskylä, Finland, June 1997, LNCS 1241, pp. 220-242.
17. Kifer, M., Kim, W., Sagiv, Y.: Querying Object-Oriented Databases. SIGMOD Conference 1992, pp. 393-402.
18. Kim, H., Park, S.: Two Version Concurrency Control Algorithm with Query Locking for Decision Support. ER Workshops 1998, pp.157-168.
19. Lämmel, R., Visser, E., Visser, J.: Strategic Programming Meets Adaptive Programming. In Proceedings of AOSD'03, Boston, USA, March 2003, ACM Press, pp. 168-177.
20. Lieberherr, K., Lorenz, D.: Coupling Aspect-Oriented and Adaptive Programming. In Filman, R. E., Erlad, T., Clarke, S., Aksit, M.. (eds.) Aspect-oriented Software Development, Addison-Wesley, 2005, pp. 145-164.
21. Lieberherr, K., Patt-Shamir, B., Orleans, D.: Traversals of Object Structures: Specification and Efficient Implementation. In ACM TOPLAS 2004, pp. 370-412.
22. Lin, W., Nolte, J.: Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking. VLDB 1983, pp. 109-119.
23. Masuhara, H., Kawauchi, K.: Dataflow pointcut in aspect-oriented programming. In 1st Asian Sym. on Programming Languages and Systems, 2003, LNCS vol. 2895, pp.105-121.
24. Mohan, C., Haderle, D.: Algorithms for Flexible Space Management in Transaction Systems Supporting Fine-Granularity Locking. EDBT 1994, pp. 131-144.
25. Mohan, C., Pirahesh, H., Lorie, R.: Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. SIGMOD Conf '92: 124-133.
26. Orleans, D., Lieberherr, K.: DJ: Dynamic Adaptive Programming in Java. The 3rd Intel Conf on Metalevel Arch. and Separation of Crosscutting Concerns, Sept. 2001, pp.73-80.
27. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In proceedings of ECOOP '05 , Glasgow, UK, July 2005, LNCS 3586, pp. 214-240.
28. Panagos, E., Biliris, A., Jagadish, H., Rastogi, R.: Fine-granularity Locking and Client-Based Logging for Distributed Architectures. EDBT 1996, pp. 388-402.
29. Popovici, A., Alonso, G., Gross, T.: Spontaneous Container Services. In Proceedings of ECOOP'03, Darmstadt, Germany, July 2003, LNCS 2743, pp. 29-53.
30. Ries, D., Stonebraker, M.: Effects of Locking Granularity in a Database Management System. ACM Transaction of Database Systems, 1977, 2(3), pp. 233-246.
31. Sakurai, K., Masuhara, H., Ubayashi, N.; Matsuura, S.; Komiya, S.: Association aspects. Proceedings of AOSD'04, March 2004, Lancaster, UK, ACM Press, pp.16-25.
32. Stein, D., Hanenberg, S., Unland, R.: Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design. In Proc. of AOSD'06, Bonn, Germany, March 2006, ACM Press, pp. 15-26.
33. Tanter, E., Gybels, K., Denker, M., Bergel, A.: Context-Aware Aspects. Software Composition (co-located with ETAPS'06), Vienna, Austria, March 2006, LNCS, pp.227-242.
34. Van den Bussche, J., Vossen, G.: An Extension of Path Expressions to Simplify Navigation in Object-Oriented Queries. In Proc. of DOOD'93, 1993, pp. 276-282.
35. Vanderperren, W., Suvée, D., Cibrán, M. A., De Fraine, B.: Stateful aspects in JAsCo. In Software Composition (at ETAPS), Edinburgh, Scotland, April 2005, LNCS, pp. 167-181.
36. Walker, R., Viggers, K.: Implementing protocols via declarative event patterns. In ACM SIGSOFT Intel. Sym. on Foundations of Soft. Eng., 2004, FSE-12, pp. 159–169.