Path Expression Pointcuts: Abstracting over Non-Local Object Relationships in Aspect-Oriented Languages

Mohammed Al-Mansari, Stefan Hanenberg

Institute for Computer Science and Business Information Systems
University of Duisburg-Essen, Campus Essen
Schützenbahn 70, 45117 Essen
Mohammed.Al-Mansari@icb.uni-due.de
Stefan.Hanenberg@icb.uni-due.de

Abstract: In aspect-oriented programming, aspects require access to join point information for selecting join points within pointcuts as well as for specifying aspect-specific behavior at such join points within advice. Unfortunately, aspect-oriented systems typically provide only local information about join points, i.e. information that is directly accessible from the execution context at the corresponding join points like the target object within a method call. However, there are many situations where the needed information is not directly available and relies on object information that is non-local concerning the execution context at the corresponding join points. As a consequence, developers are forced to specify a number of work-arounds – pointcuts and advice that neither reflect on the conceptual join point selection nor purely on the conceptual aspect behavior. In this paper, we show recurring situations in which "local" join point information is not sufficient for specifying aspects. We propose so called "path expression pointcuts" that permit to abstract over (non-local) object-relationships within pointcuts – and show that this overcomes the problem.

1 Introduction

One of the main contributions of aspect-oriented programming (AOP, [15]) is the notion of *join points*. Join points are principled points in the execution of a program [14] that can be selected by so-called pointcuts and adapted by so-called advice. Pointcuts specify selection criteria on join points based on available *join point properties* (cf. [11, 8]) – join point information that is provided by the underlying aspect-oriented system. Based on the selection of a join point, a corresponding advice often requires access to some properties of this join point in order to perform some calculations on it. Thereto, aspect-oriented languages typically provide a mechanism called *context exposure* which transfers objects addressed by pointcuts to the advice by binding such objects to explicitly defined variables within the advice (see [12]). However, not all join point properties can be exposed by using context exposure mechanism. For example, in AspectJ signature patterns cannot be exposed to the advice.

There are different kinds of join point properties [8]. For example, systems like AspectJ [14] provide for method call join points properties like *target object*, *method name* and actual *parameters*. These join point properties have in common that they can be directly derived from the local execution context of the corresponding expression representing the method call. Consequently, because of this local availability we call corresponding properties *local join point properties* [11, 8]. Since access to local join point properties can be relatively easily implemented aspect-oriented systems typically provide a large variety of such local properties.

However, aspect-oriented systems are quite miserly with properties that cannot directly be derived from the local execution context of join points. We call such properties *non-local properties* [11, 8]. One example of such a non-local property is the call-stack information. While in Smalltalk this information is locally available, it is a non-local information in other programming languages such as Java and C++. In the later case, the call-stack information is utilized for example in AspectJ by means of the *cflow* pointcut. With the aid of cflow, it is possible to specify criteria on the call stack like, for example, the occurrence of a certain actual type as target of a method call on the stack. In contrast to local properties, the provision of non-local join point properties is typically much more complex: The aspect-oriented system's task is to collect some data at runtime in order to use it at a later point in time for evaluating pointcuts. Due to this additional effort, non-local properties are rather rarely provided. In fact, most aspect-oriented systems provide only call-stack information as non-local join point properties.

One intention of aspect-oriented systems is to provide pointcut languages that permit the developer to specify *expressive pointcuts* [21] where the join point selections correspond to the developer's *mental model* [25]. This also implies, that the available join point properties provided by an aspect-oriented system must suffice the developers' needs.

However, it turns out that there are situations where it is desirable to have constructs within the pointcut language that rely on non-local join point properties. Especially, there is a large number of situations where pointcuts are needed that refer to object-relationships that cannot be calculated based on a join point's local context. As a consequence, developers cannot directly specify the mental model of join point selection. Rather, they are forced to implement a number of work-arounds that tend not only to be cumbersome and error prone, but also to mix the semantics of the join point selection with the advice code.

This paper addresses the problem of pointcuts that refer to non-local join point properties relying on *object-relationships*. Thereto, the problem is explained in more detail by an example. To overcome this problem, we propose a new language construct (path expression pointcuts) that permits to specify constraints on the object graph.

The rest of the paper is organized as follows. In section 2, we illustrate the problem by means of an example. We describe the concept of path expression pointcut in section 3 and present its semantics. Then, we show how this new construct could be used to solve the motivating problem. Section 4 discusses some related work. After some discussion presented in section 5, we conclude the paper in section 6.

2 Motivation

The motivating example comes from the domain of object persistency. Our intention is to provide object persistency as an ad-hoc functionality similar to the idea of *spontaneous containers* [22]. The idea is to provide a class PersistedList whose instances persist all contained elements. The elements that are to be added, do not need to be prepared for becoming persistent, i.e. persistency is not preplanned for the objects added to the list.

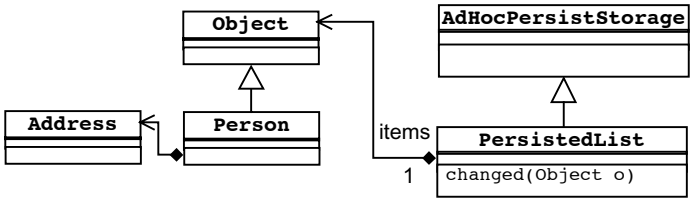


Figure 1: UML design for the problem structure.

Figure 1 illustrates an extraction from a class hierarchy making use of a persistent list. A PersistedList has a uni-directional one-to-many relationship to Object which is the system's root class. Furthermore, there is a class Person (extending Object) in the system that might be potentially added to a PersistedList. Each person object has an association to an Address object.

From the aspect-oriented point of view, it is desirable to write an aspect that handles the persistency issue for PersistedList. Hence, it is the aspect's task to notify the corresponding PersistedList whenever the state of one of its contained objects changes to ensure that the new state is stored in the persistent storage.

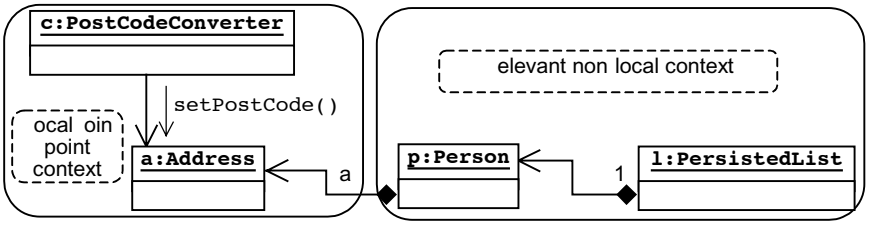


Figure 2: Person's postCode being changed by PostCodeConverter client.

Figure 2 describes a possible collaboration of objects from the previous class definitions. Furthermore, there is a PostCodeConverter object that changes for some reasons the postal code of person objects. This set operation represents the join point at which the persistedList must be notified that the owning person object has changed. From this perspective, it is desirable to specify a pointcut "*state change of an object owned by an object within a persisted list*" which exposes the owning object (the person) and the persisted list to an advice. Then, the advice sends the persisted list a message telling, that the person object has changed.

In Figure 2, there are two regions of relevant context to the change join point. The left area contains local information of the join point. This area contains the context that is provided by current AO systems, e.g., in AspectJ, `this` and `target` pointcuts are used to expose the objects `c` and `a` respectively. The right area represents the relevant non-local join point context. The non-local property in this case is the "referencing" objects, that is the corresponding persistent list `l` and the person `p`. The person object which owns the address `a` being changed should be known because it needs to be communicated to the persisted list. Also, the list itself must be accessed since it is the receiver of the message telling that the person has changed.

AspectJ provides only control flow information as non-local properties. Hence, neither the person object nor the persisted list object can be considered for join point selection or for join point adaptation in the example. Hence, it is not directly possible to express the underlying conceptual selection and adaptation in terms of pointcut and advice. Instead, the developer needs to implement some workaround for it.

The code in Figure 3 illustrates one possible workaround where the aspect `PersistedListsAspect` collects all persisted lists as soon as they are instantiated (pointcut `persistentList` and the corresponding advice). Furthermore, whenever an object in the application changes its state (pointcut `stateChange`) the corresponding advice traverses all persistent lists in order to check whether the changing object is reachable from a persistent list. Such code relies on the introspective facilities of Java. In case the changed object is reachable, the aspect sends a message `changed` to the corresponding list with the owning object as a parameter.

```
public aspect PersistListsAspect {
  WeakHashMap map = new WeakHashMap();
  // storing all persistedLists into a local data structure
  pointcut persistentLists(PersistedList list):
    execution(new(..)) && target(list);
  after returning(PersistedList list): persistentLists (list) {
    map.put(list, list);
  }
  pointcut stateChange(Object o): set(*.*) && target(o);
  after returning(Object o): stateChange(o)
  { ...
     traverse all lists to determine whether o is referenced (directly or indirectly) by a persisted list
    in such a case determine the object owning the changing object and send PersistedList a message
     changed(owningObject)
   ... }
}
```

Figure 3: AspectJ solution - exposing non-local context by additional code.

Although this solution is technically correct, it suffers from a serious problem: Neither the pointcut nor the advice represents the mental model underlying the join point selection. The join points the developer is interested in are the state changes. However, the pointcut `stateChange` does not reflect on the idea that only those join points should be selected where the changing object is reachable from a persistent list. In fact, this selection logic is specified within the corresponding advice itself – the separation of join

point selection and join point adaptation is no longer given. Furthermore, the pointcut `persistentLists` and the corresponding advice are not directly related to the pointcut `stateChange`. Instead, they are only responsible for providing the corresponding data structure that can be used within the advice belonging to `stateChange`. Consequently, understanding and maintaining the aspect is not that easy, because the mental model of join point selection and adaptation (which conceptually just consists of a single pointcut and a single advice) is hidden by its implementation.

As illustrated, the join point selection mechanism requires the availability of the join point property "*referenced by a persisted list*" at every state change join point of a person object. Unfortunately, this property could not be derived from the local information of those join points, and as a consequence, it could not be used for join point selection. This is due to the fact that in current pointcut languages the context exposed is mostly local. The only exception is the `cflow` pointcut in AspectJ that provides non-local information about call stacks. Because of this restriction to local join point properties, aspect-developers are forced to specify pointcuts and advice that do no longer reflect on the conceptual model they have in mind which makes it hard to understand and maintain the aspect.

3 Path Expression Pointcuts

This section introduces a proposal of an extension to current pointcut languages. We utilize the well-known path expression technique in a new pointcut construct called *path expression pointcuts* that permits to specify constraints on the object graph. Accordingly, developers can abstract over objects, the actual types of these objects and their relationships. Another contribution of the proposal is the adapted semantics of context exposure and advice execution mechanisms in the presence of path expressions. We implemented path expression pointcuts in our prototype language called "PurityA" [23]. Although, we use AspectJ in our examples for illustrative purposes.

Section 3.1 discusses the syntax and the semantics of the path pointcut. In section 3.2 we show how this new pointcut relates to context exposure in a language like AspectJ. In section 3.3 we explain how an adapted advice execution mechanism based on parameter bindings and the ordering of those executions. Finally, in section 3.4 we revisit the motivating example showing how path expression pointcuts solve the problem.

3.1 Syntax and Semantic

Path expressions [3], are a well-known technique used for synchronization of operation calls on data objects. They specify how threads are allowed to perform a sequence of message sends on a given object. We utilize the technique in this work such that a path expression is applied to object paths in the object graph.

The path pointcut picks out only those join points where a path described by a PathExpressionPattern from one object to another object exists in the object graph. Its general form is **path**(PathExpressionPattern) .

```
PathExpressionPattern ::= (ObjectPattern "—"
                           FieldPattern "->" )*
                          ObjectPattern.
ObjectPattern         ::= (TypePattern Identifier) |
                          (Identifier).
FieldPattern          ::= IdPattern | "/".
IdPattern             ::= ("*" [IdPattern]) |
                          (Identifier "*" [IdPattern]).
TypePattern           ::= definition of type patterns according to AspectJ syntax
Identifier            ::= definition of Identifiers according to Java syntax
```

Figure 4: PathExpressionPattern syntax.

Figure 4 depicts the syntax of PathExpressionPattern. A pattern consists of a list of object patterns and field patterns separated by the tokens "—" and "->". According to the syntax of AspectJ a FieldPattern might include *lexical abstractions*, i.e. it permits to abstract from concrete names using the wildcard "*". Furthermore, a field pattern can consist only of the token "/ ". The path expression pointcut can be directly added to the AspectJ syntax by declaring path expressions as pointcuts. Consequently, this permits to compose path expressions using the operators "&&","| | ", and "! ".

A path expression describes object paths in the (directed) object graph in an application from a source object to a target object passing a number of intermediate objects. Furthermore, the fields representing the object relationships can be specified within the path expression.

Each participating object within a path is described in terms of an object pattern which specifies the runtime type of an object and declares a name for the object that can be later used for the purpose of context exposure (which will be explained in section 3.2). According to AspectJ, it is possible to specify a lexical abstraction over type names using the wildcards "*" and ". . " (cf. [2, 16]).

The field pattern between an object o1 and an object o2 describes the fieldname over which o2 must be accessible from o1. For example, a path pattern "o1 – f1 -> o2" describes a selection criterion where object o2 must be accessible from object o1 via the field named f1. Likewise to type patterns, field patterns also provide lexical abstractions via the wildcard "*".

A special construct for field patterns is the operator "/ ". This operator permits to specify an indirect relationship between an object o1 and o2. For example, a path pattern "o1 -/ -> o2" describes a selection criterion where object o2 must be reachable from object o1 over a path of any length.

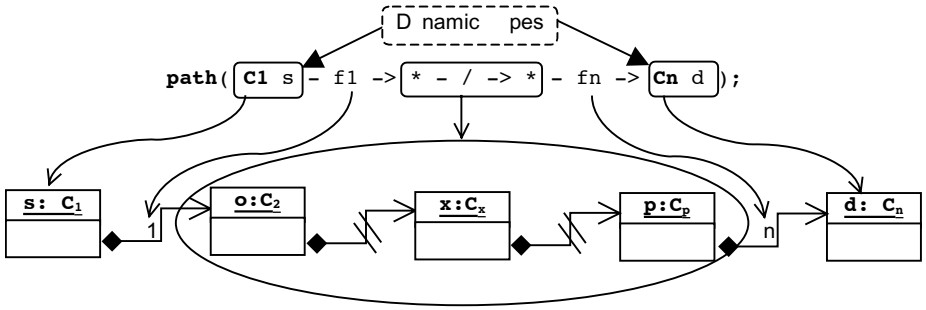


Figure 5: Exemplary path expression and corresponding matching object structure.

An exemplary path expression pointcut is illustrated in Figure 5. This pointcut selects all join points where there exists at least one path in the object graph from the source object "s" of a dynamic type "C1" to the destination object "d" of dynamic type "Cn". The first edge in each path must be an association between "s" and the object represented by the field "f1" in "s", say e.g. "o". Similarly, the last edge should be an association between an object that has a field named "fn", say e.g. "p" and the destination object "d". The circle at the center of the path corresponds to any number (≥ 0) of possible inner objects (nodes) which are specified by the wildcard "/ ".[1]

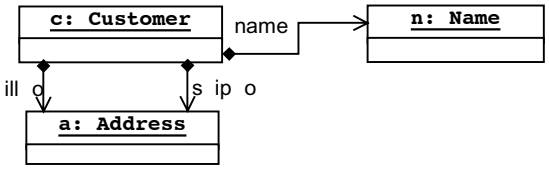
3.2 Context Exposure

One of the most important features of pointcuts in languages such as AspectJ is the context exposure mechanism [12]. Context exposure in AspectJ permits to pass objects from a pointcut described by the dynamic pointcuts this, target or args to a pointcut's header and then to an advice's header. This decouples the join point selection from the advice and permits different pieces of advice to access join point properties in a unique way. As we mentioned before, some kinds of join point properties, e.g., signature patterns in AspectJ, cannot be exposed by the context exposure mechanism.

Objects described within a path expression can be directly bound to a pointcut variable (and exposed in that way). Context exposure in the presence of path expressions works similar to context exposure in AspectJ. First, all object identifiers being used within a path expression are bound to the same identifiers declared within a pointcut's header. Since a pointcut's header already declares a dynamic type for objects described by a dynamic pointcut (corresponding to the definition of this, target or args) path expressions also permit to do so: If a pointcut header already specifies a dynamic type on an object declared within a path expression, it is allowed to leave out the type pattern within the object pattern. In case there is a type specified within the header as well as within the path expression, the combination of both type information is being used as the actual type. For example, if within the header a type A for variable x is defined (A subtypes Object) and within the path expression the type for x is declared to be B

[1] The indirect relationship between object o to x and x to p corresponds to the corresponding visialization of Join Point Designation Diagrams [26, 25].

(whereby B subtypes A), the dynamic type is B. Corresponding, if within the header a type B for variable x is defined and within the path expression the type for x is declared to be A, the dynamic type is also B.




```
1. pointcut p1(Customer c, Address a): path(c - * -> a);
2. pointcut p2(Customer c, Address a): path(c - billTo -> a);
3. pointcut p3(Customer c): path(c - * -> *) && set(* *) && target(c);
```

Figure 6: Different situations for context exposure.

Figure 6 illustrates a number of path expressions and a corresponding object graph. In the pointcut p1, objects c and a in the path expression are exposed and made available for the selection and adaptation mechanisms. The number of matching paths is 2, both are from c to a and of length 1. The first path is via the field billTo and the other one via the field shipTo. Likewise, the path expression in pointcut p2 matches one path of length 1 from c to a via the field billTo. According to both pointcuts, objects c and a are exposed. Finally, there are three matching paths of length 1 according to the path expression pointcut in p3. These paths are: c – name -> n, c – billTo -> a, and c – shipTo -> a, where only object c needs to be exposed.

One interesting observation from the definition of pointcut p3 in the above example is that the variable c has been used twice: In the source of the path expression and in the target pointcut. This is not valid in AspectJ, however, in the presence of path expression pointcuts this is permitted since there is a means to unify all these occurances to refer to the same corresponding exposed object.

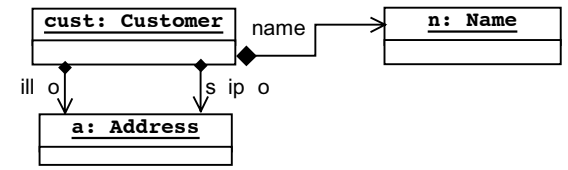
3.3 Advice Execution, Parameter Binding and Ordering

One consequence of path expressions is that it is possible (or rather the usual case) that a pointcut containing a path expression provides a number of different results as valid parameter bindings in the pointcut.

Figure 7 illustrates such a situation. The pointcut pc selects all setter join points where the target object is an instance of Customer. Furthermore, it binds the changing customer object to the variable c and all member objects to the variable o. The customer refers to two objects (a Name instance and an Address instance) via three different paths: cust – name -> n, cust – billTo -> a, and cust – shipTo -> a.

However, only the distinct valid variable bindings in the pointcut are considered. Hence, in the above example, there are two valid bindings of the variables c and o: One binding (c=cust, o=n) and one binding (c=cust, o=a).

Because of that, a new semantics for advice execution is necessary that differs widely from the current semantics of advice execution in AspectJ. According to the example above, each associated advice to the pointcut `pc` must run as many times as there are valid parameter bindings[2]. Therefore, an explicit ordering of pointcuts might become important and developers require means to specify the execution order of advice.




```
pointcut pc(Customer c, Object o):
  path(c - * -> o) && set(* *) && target(c);
```

Parameter Bindings: `(cust, a)` and `(cust,n)`

Figure 7: Multiple bindings of pointcut parameters.

In our model, we have implemented an additional `orderBy` clause that can be associated with a pointcut in order to apply advice executions in a specific order. The `orderBy` clause has a target object and method name as a parameter. The corresponding method specifies how the parameter bindings are to be ordered. If no `orderBy` clause is specified, the parameter bindings are unordered.

Developers have to specify an ordering method on their own similar to the method `compare` as specified in the J2SE in type `Comparable`. An ordering method is a method that has two parameters of type array of `Object`. Each array represents a single variable binding. The return type of the ordering method is boolean: The method returns `true`, if the first parameter binding should appear before the second one, otherwise the method returns `false`.

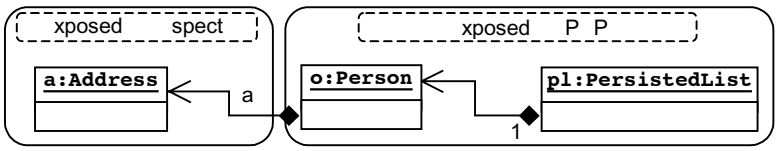
```
public boolean addressChanged(Object[] o1, Object[] o2) {
  Address a1 = (Address) o1[1]; Address a2 = (Address) o2[1];
  return a1.getPostCode() < a2.getPostCode();
}

pointcut addressChanged(Person p, Address a):
  set(* *) && target(a) &&
  && path(p -*-> Address a) orderBy(this.addressChanged);
```

Figure 8: Use of `orderBy` in pointcuts.

Figure 8 illustrates the use of an ordering clause. The clause `orderBy` specified at the end of pointcut `addressChanged` has `this.addressChanged` as parameter. Hence, there must be a method `addressChanged` defined in the surrounding type that has two parameters of type `Object[]`. Figure 8 illustrates a possible implementation of `addressChanged`. Within the body, the second element of `o1` and `o2` are assigned to a variable of type `Address`. Then, the postal codes of both address objects are compared

[2] Such an approach of advice execution has been already proposed in other pointcut languages (cf. e.g. [24]).

and the lesser postal codes are considered to appear first. Consequently, all multiple executions of advice referring to pointcut addressChanged will have parameter binding where the first bindings have smaller postal codes within their address objects than the latter ones.




```
pointcut changePerson(Address a, Object o, PersistedList pl):
  set(* *.a) && target(a) && path(pl -*-> o -/-> a);
after(Address a, Object o, PersistedList pl): changePerson(a, o, pl) {
   pl.changed(o);
}
```

Figure 9: All relevant JPP's are made accessible from the join point local context.

The reason for providing such ordering schema is to give developers the ability to specify their own ordering rules based on the resulting parameter bindings in a more fine-grained manner. This is due to that the competing executions are on the same advice but not between different pieces of advice from different aspects. Moreover, each pointcut that includes a path expression may require its own ordering criteria. Within the associated ordering method, the developer can use the exposed objects to reach any other referenced object that she/he wants to use for ordering. Note that a single ordering method can be used by more than one pointcuts.

3.4 Motivating Example Revisited

Figure 9 shows how path expression pointcuts could be used to provide locality to the needed join point properties, which in turn, preserves the join point selection semantics. From the figure, any join point is selected if there is a field setting on the address object a that is a part of the person object, which is in turn accessible from a persisted list pl. Objects pl and o are exposed and bound by the path pointcut while the target pointcut exposes the address object a.

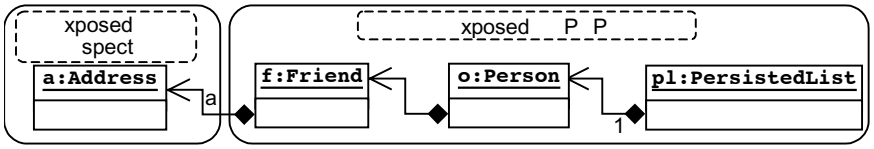


Figure 10: Indirect relationship between o and a accessible by PEP.

Note that the path expression pattern uses the wildcard "/ " to indicate that there might be any number of intermediate objects between the person object o and the address object a. For example, consider the diagram in figure 10, where the person object is

associated with the object f of type Friend that is, in turn, associated with the address object a. The path expression in the pointcut changePerson in figure 9 still holds in this case since object f is an inner object in the path between o and a.

This solutions illustrates how path pointcuts provide access to the relevant non-local information to the join points. At the same time, the pointcut specifies the object relationships along the path from p1 to a through o which exactly reflects the developer's mental model of the corresponding join point selection. Moreover, the advice specifies only the message to the persisted list and does not contain any code representing parts of a join point selection. Hence, there is a strict separation of join point selection and adaptation in the example as desired by the underlying mental model.

4 Related Work

Path Expressions
Path expressions, first introduced in [3], are used in the synchronization of operations on data objects. In other words, they specify how threads are allowed to perform a sequence of message sends on a given object. Path expressions are heavily used in object-oriented query languages such as EJB-QL [1] and [5, 27]. In our work, we studied the benefit of the application of path expression technique on AO systems and how it increases the expressiveness of pointcut languages.

Expressive Pointcuts and Non-local Information
In [21], the authors discuss the need for expressive pointcuts in order to increase the modularity. Their argument is based on a comparison between an object-oriented and an aspect-oriented solution for the observer pattern [6]. Their observation is that current pointcut languages are not sufficient with respect to absorbing changes. In their prototype language, ALPHA, pointcuts are Prolog queries over a database of a number of different semantic models of the program. This meets with our work that we, too, make the object graph accessible during the execution of the program. ALPHA requires a significant large database since it may store almost all program models. In contrast, path expression pointcuts needs only the heap to be stored.

The need for dynamic information non-local to the join points is stated also in [28, 4]. Stateful aspects are aspects that define triggering conditions based on finite state transitions. Those conditions are used by the advice to trigger their executions on a protocol sequence of join points. However, the kind of join point property needed here is similar to call stack information, which differs from "referenced by" information, which represents relationships between objects rather than protocol or sequences of events. Some other works already show the need for more pointcut constructs based on non-local join point properties. For example, data flow pointcuts [19], solved the problem of non-locality of data flow information join point property.

Adaptive Programming
In adaptive programming (AP), the program changes its behavior based on the current state of its environment [7]. DJ [20] is a pure Java library that provides class graphs, traversal strategies and adaptive visitors to traverse objects of an application. The traversal strategy is a path expression where traversals are specified using the "*from-to*" expression and when needed "*through and bypassing*". In contrast to path expression pointcuts, AP is not applied on pointcut language constructs.

Strategic Programming
Strategic Programming promises to provide the programmer with full traversal control [17]. SP traversals resulted from applying traversal schemes with basic data processing computations as arguments. To define object-oriented strategies, the strategic programmer has to modify the visitor objects and extend them by his/her strategy. Beside this overhead, the strategies are applied to tree-shaped data, and complying the object structures, which are mostly graph-based, needs hard compromises. StrategoAspect [13], is being developed to combine AO and SP.

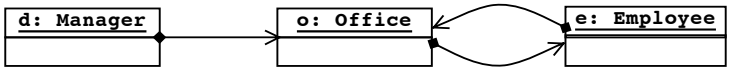
Association Aspects
Sakurai et. al. [24] introduce a new pointcut designator to AspectJ, namely `associate`, which is used to associate extended per-object aspect instances to a group of objects. These associations are used at runtime to implement crosscutting concerns that have stateful behavior related to a particular group of objects, however, the associations should be determined upfront which is opposite to the intention of path expression pointcuts. Moreover, the authors did not specify the ordering of the multiple executions of the advice in the associated aspect instances.

Morphing Aspects
There are other attempts to provide abstraction on object relationships for the purpose of join point selection. Morphing aspects [10] are proposed to solve the problem unnecessary join point checks caused by complete weaving feature of current aspect-oriented languages. Also, specifying join points based on non-local object relationships is presented in [9]. These works strength our argument that pointcut languages must provide a means to quantify on join points based on object relationships, however, these solutions make use of reflective features of the underlying programming languages and not by means of pointcut constructs.

5 Discussion

While we consider path expressions to be good abstractions for pointcut languages in aspect-oriented systems there are still a number of issues that need to be discussed in more detail.

First of all, there seems to be some agreement in the aspect-oriented community that there is a need to guarantee the termination of the evaluation of pointcuts. Hence, each new pointcut language construct should be checked whether it fulfills this characteristic. Path expression pointcut fulfills this requirement due to how variable binding is defined. A typical problem of non-determination is the existence of cycles in the path that is compared against the path expression. Figure 11 illustrate an object graph that has a cycle between the objects `o` and `e`. Consequently, there is an infinite number of paths in this graph. However, the defined variable binding as explained in section 3.3 requires only checking all possible bindings for the defined variables within the path expression. This guarantees the termination of a path expression pointcut for a given path. Figure 11 contains an exemplary application of path expression pointcut with the variable `x`, `y` and `z`. Since variable `x` is bound to the actual type `Manager`, all resulting bindings have the object `d` bound to the variable `x`. For the variables `y` and `z`, all combinations of `o` and `e` are valid variable bindings. Hence, the variable bindings of this pointcut are finite, although there is an infinite number of paths.




```
pointcut pc(Manager x, Object y, Object z): path(x -/-> y -/-> z)…
```

Parameter bindings: `(x=d, y=o, z=e), (x=d, y=o, z=o), (x=d, y=e, z=o),` and `(x=d, y=e, z=e)`

Figure 11: Cycles in object graphs.

However, since the here proposed approach also permits developers to specify an ordering of variable bindings using ordinary methods of the base language, and since the base language is Turing-complete, the definitions of ordering methods do not guarantee the termination characteristic. Although this could be considered to be a weakness in the approach, we still rely on this approach for two reasons. First, defining an ordering method can be considered rather as a trivial task. Hence, we do not think that the developer is overstrained with a definition of a terminating ordering method. Second, until now, we were not able to determine common abstractions of what ordering schemes are typically desired by the developer (like for example forward or backward rules like the one used in [18]). Consequently, we did not provide special language constructs for this task.

One further issue with path expression pointcuts is that their expressiveness is probably more restricted than needed in order to guarantee the termination. However, our current experiences with path expression pointcuts did not reveal any examples, where a more expressive construct for reasoning on the object graph is required.

One typical problem with the evaluation of path expressions is the performance problem. In general, a path expression pointcut potentially requires the traversal of a large object graph which is a time consuming task. In our current implementation of path expressions, we do not provide any static analysis in order to exclude already paths that

can never match a given path expression. A first possible approach could be to utilize static type information of type hierarchies in order to reduce the number of potentially matching paths. This would correspond to the static analysis techniques for join point shadows as applied in AspectJ (cf. [12]).

How far such problems become relevant and how far the potential solutions are feasible for path expressions depend on how path expressions will be used. The main problem here is to determine how often and in what situations path expressions will be used in order to approximate the caused runtime overhead. Unfortunately, we have until now no overview how the existence of path expressions potentially influences the specification of aspects in the future. Hence, we cannot come until now to any conclusion about the performance problems.

6 Conclusion

This paper focuses on the locality issues of join point properties that are based on object relationships in aspect-oriented systems. We illustrated the importance for the non-local properties to be accessible from join points execution context for selecting as well as adapting those join points. We used an example from the object persistence domain to show how current aspect-oriented approaches fail to make relevant non-local information accessible from the aspects by means of the available pointcut constructs.

As a solution to this problem, we proposed path expression pointcuts that selects join points based on given path expression patterns. Path pointcuts select the join points when there exists at least one path that matches the given path expression pattern. Developers are able to use the proposed pointcut to make non-local relevant information accessible from the pointcuts and the associated advice declarations. This is achieved by the help of the context exposure mechanism of the path expression pointcuts.

We discussed the possible technical issues that should be addressed carefully in any implementation of path expression pointcuts and we presented some possible solutions to those limitations. These issues are going to be our main focus for the future work. Another important work direction that we are concerning also is completing the formal semantics of the path expression pointcut.

Although we are currently only aware of a number of examples how path expressions help specifying aspects, we are not completely aware of the impact path expressions might have on aspect-oriented systems in the future. However, our current studies have shown that path expressions are good abstractions that permit developers to specify their mental model of join point selection and join point adaptation in a more adequate way than provided by current aspect-oriented systems.

References

[1] Adatia, R. et al.: Professional EJB – Wrox Press, ISBN 1-861005-08-3, 2001.

[2] AspectJ homepage: http://eclipse/org/aspectj.
[3] Campbell, R.; Habermann, A.: The Specification of Process Synchronization by Path Expressions. Lecture Notes in Computer Science (Editor G. Goos and J. IIartmanls), pp. 89-102, V16, Springer Verlag, 1974.
[4] Douence, R.; Fradet, P.; Südholt, M.: Composition, Reuse and Interaction Analysis of Stateful Aspects. In Proceedings. of AOSD'04: 141-150, Lancaster, UK, March 2004.
[5] Frohn, J.; Lausen, G.; Uphoff, H.: Access to Objects by Path Expressions and Rules. I Proc. Of VLDB'94, pages 273-284; Santiago, Chile, 1994.
[6] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. AddisonWesley, 1995.
[7] Gouda, M. G.; Herman, T.: Adaptive programming. IEEE Transactions on Software Engineering, vol. 17, no. 9, pages 911-921, September 1991.
[8] Hanenberg, S.: Design Dimensions of Aspect-Oriented Systems. PhD dissertation. Duisburg-Essen University, January 2006.
[9] Hanenberg, S.; Hirschfeld, R.; Unland, R.: Aspect Weaving: Using the Base Language's Introspective Facilities to Determine Join Points. Workshop on Advancing the State-of-the-Art in Runtime Inspection at ECOOP, Darmstadt, Germany, July 21, 2003.
[10] Hanenberg, S.; Hirschfeld, R.; Unland, R.: Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving. In Proceedings of AOSD'04; Lancaster, UK; March 22-26; ACM-Press, pp. 46-55; 2004.
[11] Hanenberg, S.; Stein, D.; Unland, R.: Eine Taxonomie für aspektorientierte Systeme. In Proc. of Software Engineering 2005, Essen, Germany, March, 8-11, 2005, LNI P-64, pp. 167-178.
[12] Hilsdale, E.; Hugunin, J.: Advice weaving in AspectJ. In Proceedings. of AOSD'04, p. 26-35. ACM Press, 2004.
[13] Kalleberg, K.; Visser, E.: Combining Aspect-Oriented and Strategic Programming. In H. Cirstea and N. Marti-Oliet, editors, Workshop on Rule-Based Programming (RULE'05), Electr. Notes Theor. Comput. Sci. 147(1): 5-30, Nara, Japan, April 2005.
[14] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: An overview of AspectJ. In Proceedings of ECOOP '01, p. 327-353, Budapest, Hungary, June 18-22, 2001.
[15] Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J.: Aspect-oriented programming. In Proceedings of ECOOP'97, volume 1241, pages 220-242. SpringerVerlag, 1997.
[16] Laddad, R.: AspectJ in Action - Practical Aspect-Oriented Programming, Manning Pub. Co. 2003, ISBN: 1930110936.
[17] Lämmel, R.; Visser, E.; Visser, J.: Strategic Programming Meets Adaptive Programming. In Proceedings of AOSD'03, pages: 168-177, Boston, USA, March 2003.
[18] Lieberherr, K.; Patt-Shamir, B.; Orleans, D.: Traversals of Object Structures: Specification and Efficient Implementation. In ACM TOPLAS 2004, pages 370-412.
[19] Masuhara, H.; Kawauchi, K.: Dataflow pointcut in aspect-oriented programming. In 1st Asian Symposium on Programming Languages and Systems, LNCS-vol 2895, pages 105-121, 2003.
[20] Orleans, D.; Lieberherr, K.: DJ: Dynamic Adaptive Programming in Java. Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, p.73-80, September 25-28, 2001.
[21] Ostermann, K.; Mezini, M.; Bockisch, C.: Expressive pointcuts for increased modularity. In proceedings of ECOOP '05, pages: 214-240, 2005.
[22] Popovici, A.; Alonso, G.; Gross, T.: Spontaneous Container Services. In Proceedings of ECOOP'03, pages: 29-53, Darmstadt, Germany, July 2003.
[23] PurityA project homepage: http://dawis.icb.uni-due.de/research/aosd/puritya/.
[24] Sakurai, K.; Masuhara, H.; Ubayashi, N.; Matsuura, S.; Komiya, S.: Association aspects. Proceedings of AOSD'04, p.16-25, March 22-24, 2004, Lancaster, UK.
[25] Stein, D.; Hanenberg, S.; Unland, R.: Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design. In Proceedings. of AOSD'06, pages: 15-26, Bonn, Germany, March 20-24, 2006, ACM Press.

[26] Stein, D.; Hanenberg, S.; Unland, R.: Query Models; in: Baar, Th., Strohmeier, A., Moreira, A., Mellor, St., Proc. of the 7th International Conference on the Unified Modeling Language (UML 2004), Lisbon, Portugal, October 11-15, 2004, Springer, LNCS 3273, pp. 98-112.
[27] Van den Bussche, J.; Vossen, G.: An Extension of Path Expressions to Simplify Navigation in Object-Oriented Queries. In Proc. of Intel. Conference on Deductive and Object-Oriented Databases (DOOD), pages 276-282, 1993.
[28] Vanderperren, W.; Suvée, D.; Cibrán, M. A.; De Fraine, B.: Stateful aspects in JAsCo. In Proceedings of SC 2005, LNCS, pages: 167-181, Edinburgh, Scotland, Apr. 2005.