Instance Pointcuts

Selecting Object Sets Based on Life-Cycle Phases

Kardelen Hatun
TRESE, University of Twente,
7500AE Enschede, The Netherlands
k.hatun@utwente.nl

Christoph Bockisch
TRESE, University of Twente,
7500AE Enschede, The Netherlands
c.m.bockisch@utwente.nl

Mehmet Aksit
TRESE, University of Twente,
7500AE Enschede, The Netherlands
m.aksit

Abstract

In the life-cycle of objects there are different phases. The phase in which an object currently is, affects how it is handled in an application; however phase shifts are often implicit. Selecting objects according to such phase shifts results in scattered and tangled code. In this study we propose a new kind of pointcut, called *instance pointcuts*, for maintaining sets that contain objects with a specified usage history. Specify are provided in terms of pointcut-like specifications selecting events in life-cycle of objects. Instance pointcuts can be reused, by refining their selection criteria, e.g., by restricting the scope of an existing instance pointcut; and they can be composed, e.g., by set operations. These features make instance pointcuts easy to evolve according to new requirements. Our approach improves modularity by providing a fine-grained mechanism and a declarative syntax to create and maintain phase-specific object sets.

Categories and Subject Descriptors D.3.1 [*Formal Definition and Theory*]: [syntax, semantics]; D.3.4 [*Processors*]: [code generation]

1. Introduction

In object-oriented programming (OOP), the encapsulated state and the provided behavior of objects is dictated by their type. Nevertheless, often objects of the same type need to be treated differently. For example, consider a security-enabled system with a type for users. The treatment of a user object depends on the user's privileges and possibly also on the past execution: Maybe we want to reduce the privileges when the user did not change the password for a while, or privileges are added or withdrawn at runtime in other ways.

Software design patterns [7] are another popular, more general example for dynamically varying the treatment of objects. Several design patterns define *roles* for objects, which can be assigned or removed at runtime, and the roles determine how an object is handled. However, while the pattern localizes the handling of object roles, the assignment of roles is usually scattered over the multiple source modules. As example, consider the *observer pattern*. To assign the role of *being observed* to an *subject*, an *observer* must be added to its observer list. The code for adding observers is generally not well localized. Other similar examples are the *adapter*, *decorator*, or *proxy* patterns.

More generally, we can say that objects have a life-cycle and we sometimes need to handle objects according to the life-cycle phase they are currently in. Often the shift from one life-cycle phase to another is implicitly marked by events, e.g., passing an object from one client to another. We claim that to improve the modularity of source code, a declarative definition of relevant object life-cycle phases is necessary. Furthermore, it must be possible to reify the set of objects that currently are in a specified life-cycle phase to consider this information when handling an object.

Throughout this paper, we will elaborate an example online shop application to demonstrate the life-cycle of objects and the need for reification. In this example "vendor" objects represent the suppliers and "product" objects represent the products they sell. Assume we would like to keep the list of products which were applied the happy-hour discount. For each product, a different time slot can be defined as the happy hour, so the list of products that are discounted is changing over time. Grouping objects according to criteria which cannot be directly accessed through programming language constructs — such as which class they were initialized in, which method they were passed to as an argument, or (as in the example) the time at which they are passed to a method, requires invasive insertion of bookkeeping code.

Aspect-oriented programming (AOP) can be applied to separate this bookkeeping code from the business logic of the program. But in AOP, *pointcuts* select sets of so-called *join points* which are points in time during the execution of the program. Current aspect-oriented languages do not sup-

port a *declarative specification* of the objects belonging to a life-cycle phase; instead an *imperative implementation*, typically following the same pattern, is required for collecting those objects.

A consequence of such an imperative solution is reduced readability and maintainability due to scattering, tangling and boilerplate code. Another issue is the lack of composition and checking mechanisms for the imperative bookkeeping. It is not possible to reuse the previously written code which results in code that is hard to maintain and hinders software evolution. Also the warnings and errors do not indicate the proper context and relevant information to guide the programmer.

To offer better support for processing objects according to their life-cycle phases, we propose a new mechanism, called *instance pointcuts*, to select sets of objects based on the events in their execution history. Instance pointcuts are used to declare the beginning and the end of a life-cycle as events. New instance pointcuts can be defined by reusing existing ones in two ways: first, by *refining* the expressions defining the relevant events and second, by *composing* two or more instance pointcuts using set operators.

An instance pointcut's concise definition consists of three parts: an identifier, a type which is the upper bound for all the selected objects in the phase-specific set, and a specification of relevant objects. For a basic instance pointcut definition, the specification utilizes *pointcut expressions* to select events that define the start and the end of life-cycle phases and to expose an object. At these events, the object is added to or removed from the set associated with the instance pointcut. We refer to this responsibility as *maintaining* the instance pointcut's object set. New instance pointcuts can be derived from existing ones. Firstly, a new instance pointcut can be derived from another one by restricting the type of selected objects. Secondly, a new instance pointcut can be created by reusing the object selection expressions of the existing ones. Lastly, instance pointcuts can be composed arbitrarily by means of set operators.

In this paper we present a prototype of instance pointcuts as an extension to AspectJ [12] and explain its semantics by explaining our compiler which transforms instance pointcuts to plain AspectJ and advanced dispatching library calls.

We reuse the term pointcut for our concept, because it provides a declarative way of specifying crosscuts. Nevertheless, the instance pointcuts select *objects* whose usage crosscuts the program execution rather than *points (or regions) in time* [13] as traditional pointcut do. Therefore, our instance pointcuts cannot immediately be advised by AspectJ advice, although we offer the possibility to advise the points in time when the extent of instance pointcuts changes (cf. Section 4.4.2).

The declarative nature of instance pointcuts gives rise to several compile-time checks which are not automatically possible with equivalent imperative code. Such checks are important to notify the developer when the instance pointcut set is guaranteed to be empty, incompatible types are used in compositions and refinements, etc. These checks help the developer to implement his concern correctly and achieve consistency in phase-specific object sets.

The rest of the paper is organized as follows, in section 2 we present a small case study and explain our motivation for the proposed approach, and we formulate a problem statement and goals for our work in section 3. In section 4, a detailed description of instance pointcuts and its various features are presented. Section 5 explains how instance pointcuts are compiled. We then present a discussion on the validation of our approach and outline possible checks in section 6. We conclude by discussing related work and giving a summary of our approach.

2. Motivation

Supporting *unanticipated* extensions introduces new implementation concerns like creating specific object sets. Objects can be grouped according to how they are used (passed as arguments to method calls, act as receiver or sender for method calls, etc.) and concerns of an application may be applicable only to objects used in a specific way. Therefore we must be able to identify and select such objects. We want to expose sets of objects belonging to the same life-cycle phase by means of a dedicated language construct such that the implementation of phase-dependent concerns can be explicit.

In Figure 1, we outline a part of the architecture of an online shop application. We use this scenario to give examples of grouping objects into sets according to how they are used and how to use these sets in the implementation of concerns.

2.1 Example Architecture

In an online shop application, objects of the same type can exist at different stages of their life-cycle. In Figure 1 the static structure of a simplified online shop is shown. This structure shows part of the system from the Vendor and the OnlineShop's perspective. Vendors can submit different kinds of Discounts (not shown in the figure) to the ProductManager for the Products they are selling. Product is the root of the type hierarchy that represents different kinds of items that are sold in the online shop. Product is parent to the classes such as BeautyProduct, SportProduct (not shown in the figure). Each Product holds a list of Discounts that are applied to it. The OnlineShop has a user interface represented by the OnlineShopUI class, which is used to display information to the customers.

2.2 Unanticipated Extensions

A new feature is added to the online shop which requires creating an alert when a product is applied a surprise discount. The list of surprise discounted products should be available to the user at any time. The surprise discounts are submitted by Vendors and they can be submitted or

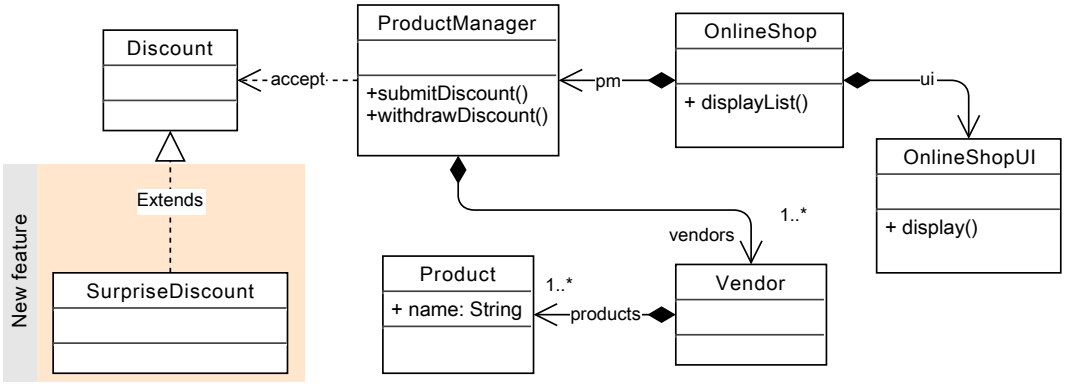


Figure 1: Part of an online shop application

withdrawn any time. In order to realize this extension in an OO-approach, we need to change several classes. First the class ProductManager should keep a set of Products to which a surprise discount is applied, in listing 1 this is shown in line 3. This set is updated when a new discount of type SurpriseDiscount is submitted or withdrawn (lines 4– 15). There should also be some changes in the OnlineShop class. A createDiscountAlert method should be added. Also the displayList method should be updated to include the surpriseDiscount list defined in the ProductManager class.

OO-solution is scattered among the classes ProductManager and OnlineShop and tangled with multiple methods.

An aspect-oriented implementation can offer a better solution by encapsulating the concern in an aspect. Listing 2 shows a possible solution. The set of products which are applied a surprise discount is kept in the aspect (line 2). The following two pointcuts submit and withdraw selects the products to which a SurpriseDiscount is applied (lines 3 – 4). The corresponding advice declarations for these pointcuts maintain the surpriseDiscount set. The submit pointcut triggers the surprise discount alert method (line 8). There is also the display pointcut (line 5), which intercepts the call to displayList method and add the condition for the surprise discount list in an around advice (lines 13 – 17). This aspect includes an inter-type declaration which adds the createDiscountAlert method to the OnlineShop class.

```
1  class ProductManager{
2    ...
3    Set<Product> surpriseDiscount = createSet();
4    public void submitDiscount(Product p, Discount d){
5      ...
6      if(d instanceof SurpriseDiscount){
7        surpriseDiscount.add(p);
8        OnlineShop.createDiscountAlert(p);
9      }
10   }
11   public boolean withdrawDiscount(Product p, Discount d){
12     ...
13     if(d instanceof SurpriseDiscount)
14       surpriseDiscount.remove(p);
15   }
16 }
17 class OnlineShop{//SINGLETON
18   ...
19   public void createDiscountAlert(Product p){
20     //create surprise discount alert for p
21   }
22   public void displayList(String listType){
23     if(listType.equals(''surprise''))
24       INSTANCE.getUI().display(ProductManager.surpriseDiscount);
25     ...
26   }
27 }
```

Listing 1: A Java implementation of discount alert concern

2.3 Discussion

AOP already helps to localize the concern and to add it without the need to modify existing code. However maintenance of the surpriseDiscount set requires the same boilerplate code as the OO solution does. Essentially the code selects Product objects based on the discount they are applied to and deselects them once they are rid of this discount. This marks a phase in the life-cycle of a Product object. None of the solutions presented in this section offer declarative means to define such a life-cycle phase. Furthermore, reusing such existing reifications of objects in a specific life-cycle phase by refining or composing them is not conveniently supported at all; e.g., if we want to find the subset of BeautyProducts of the surpriseDiscount set, we have to iterate over it and check instance types to create a new set. Such imperative definitions are difficult or impossible to analyze by the compiler. For instance, it may be desirable to warn developers about instance pointcuts that provably will never match any object, e.g., because the selection events will never happen. With a

```
1  aspect SDiscount{
2    Set<Item> surpriseDiscount = createSet();
3    pointcut submit(Product p): call(* ↙
         ProductManager.submitDiscount(..)) && args(p, ↙
         SurpriseDiscount);
4    pointcut withdraw(Product p): call(* ↙
         ProductManager.withdrawDiscount(..)) && args(p, ↙
         SurpriseDiscount);
5    pointcut display(String listType): call(* ↙
         OnlineShop.displayList(..)) && args(listType);
6    after(Product p): submit(p){
7      surpriseDiscount.add(p);
8      OnlineShop.INSTANCE().createDiscountAlert(p);
9    }
10   after(Product p):withdraw(p){
11     supriseDiscount.remove(p);
12   }
13   void around(String listType):display(listType){
14     if(listType.equals(''surprise''))
15       OnlineShop.instance().getUI().display(surpriseDiscount);
16     proceed(listType);
17   }
18   public void OnlineShop.createDiscountAlert(Product p){
19     //create surprise discount alert for a product
20   }
21 }
```

Listing 2: An AspectJ implementation of discount alert concern

more declarative notation, a compiler would be able to identify such situations.

3. Problem Statement

In the previous section we have demonstrated two things: First, the implementation of some concerns requires accessing groups of objects with similar usage history. Second, making such groups accessible to the program in a modular and re-usable way is not supported by current programming languages.

Since creating object sets according to execution events is a cross-cutting concern, we claim that a new programming technique in the style of aspect-oriented programming is required for modularizing concerns depending on object groups. Such a programming technique must satisfy the following need:

- A declarative way of selecting/de-selecting objects according to the events they participate in should be provided.
- The selected objects should be accessed as a set.
- The set of objects should be accessible and any changes to this set i.e. adding/removing objects should create a notification.
- For the same kind of objects, the sets should be composable to obtain new sets.

4. Instance Pointcuts

To support the requirements outlined in the previous section, we propose a new kind of pointcut for declaratively selecting objects based on their life-cycle phases. The beginning and the end of a phase is marked by events. An instance pointcut is a declarative language construct that is used to reify and maintain a set of objects of a specified type. The objects are selected over a period marked by events in their life-cycle. Instance pointcuts modularize the object selection concern and makes it declarative.

In the remainder of this section, we will explain instance pointcuts in detail. Instance poincuts can be implemented as an extension to arbitrary OO-based aspect-oriented languages. In this work, we have implemented a prototype as an extension to AspectJ. Therefore, the examples given throughout this paper are based on AspectJ.

A concrete instance pointcut definition consists of a left hand-side and a right-hand side (Figure 2, rule 1). At the left-hand side the pointcut's name and a type is declared. An instance pointcut does not declare pointcut parameters since it has the specific purpose of exposing one object from an event; it has a single implicit parameter called **instance** of the declared type.

At the right-hand side the instance pointcut expression selects the desired events from join points and then binds the exposed object (represented by the **instance** parameter) as a member of the instance pointcut's set.

⟨*instance pointcut*⟩ ::= '`instance pointcut`' ⟨*name*⟩ '<' ⟨*instance-type*⟩ '>' '**:**' ⟨*ip-expr*⟩ ('`UNTIL`' ⟨*ip-expr*⟩)?

⟨*ip-expr*⟩ ::= ⟨*after-event*⟩ '`||`' ⟨*before-event*⟩
| ⟨*before-event*⟩ '`||`' ⟨*after-event*⟩
| ⟨*after-event*⟩
| ⟨*before-event*⟩

⟨*after-event*⟩ ::= '`after`' '`(`'⟨*pointcut-expression*⟩'`)`'

⟨*before-event*⟩ ::= '`before`' '`(`'⟨*pointcut-expression*⟩'`)`'

Figure 2: Grammar definition for instance pointcuts

4.1 Add/Remove Expressions

For each instance pointcut events must be selected at which objects are added to its set, otherwise the instance pointcut set would always be empty. Optionally, an instance pointcut can select events at which objects are removed from its set. The 'add to set' and 'remove from set' operations are implicitly performed when certain events specified in the corresponding *sub-expression* (cf. Figure 2, rule 1) occur. The

first expression is the mandatory *add expression*. After the **UNTIL** clause an optional sub-expression called the *remove expression* can be defined.

In AspectJ join points mark *sites* of execution; a join point by itself does not define an event. Pointcut expressions select join points and pointcuts are used with advice specifications to select a particular event in that join point. As discussed by Masuhara et al. [13] such a region-in-time join-point model hinders re-use of pointcuts.

We combine pointcut expressions with advice specifiers and obtain *expression elements*. Each expression element contains a pointcut expression, which matches a set of join points. Then, from these join points, according to the advice specifier the before or after events are selected. Both add and remove expressions are composed of expression elements which can be a *before element* or an *after element* (Figure 2, rule 3-4). A sub-expression (add/remove expression) contains at least one *expression element* and at most two. In Figure 2 the second grammar rule depicts this statement.

In Figure 2 rules 3 and 4 contain the $\langle pointcut{-}expression \rangle$ rule which represents an AspectJ pointcut expression. However we have introduced a restriction that in every pointcut expression there must be *exactly* one binding predicate (**args**, **target** etc.) that bind the **instance** parameter. Furthermore, it is mandatory to bind the **instance** parameter since it represents the object to be added or removed from the set.

Allowing only the binding of one value at each event is a limitation of our current language prototype. It would be a straight-forward extension of the instance pointcut language to allow binding multiple values to the implicit **instance** parameter and then add all bound values to the instance pointcut's set. However, this would require a more complex code generation.

The binding predicates are extended to include the **returning** clause. The **returning** clause binds the value returned by a method or a constructor. In AspectJ the syntax is restricted and **returning** can only be used in an after advice, since the returned value is only available after a method finishes execution, this is also true for us. Although we do not have this restriction syntactically, we enforce that the **returning** clause is used only with the after event selector by means of a semantic check.

In an instance pointcut expression, it is only possible to *OR* a before event with an after event. The *before* clause selects the start of executing an operation (i.e., the start of a join point in AspectJ terminology) and the *after* clause selects the end of such an execution. For two operations that are executed sequentially, the end of the first and the start of the second operation are treated as two different events. Thus, the before and after clauses select from two disjoint groups of events and the conjunction of a before and an after clause will always be empty.

```
1 static instance pointcut surpriseDiscount<Product>:
2   after(call(* ProductManager.submitDiscount(..))
3   && args(instance, SurpriseDiscount))
4   UNTIL
5   after(call(* ProductManager.withdrawDiscount(..))
6   && args(instance, SurpriseDiscount));
```

Listing 3: A basic instance pointcut declaration with add and remove expressions

The instance pointcut in Listing 3 shows a basic example. The left-hand side of the instance pointcut indicates that the pointcut is called surpriseDiscount and it is interested in selecting Product objects. On the right hand side, there are two expressions separated by the **UNTIL** keyword. The first one is the add expression. It selects the join-point marked by the method submitDiscount and from the context of this event it exposes the Product object with the **args** clause and binds it to the **instance** parameter. The second one is the remove expression and it selects the after event withdrawDiscount call and exposes the Product instance in the method arguments and binds it to the **instance**. This pointcut is the solution of the set maintenance problem presented in the motivation (section 2).

Note that instance pointcuts do not keep objects alive, as instance pointcuts are non-invasive constructs, which do not affect the program execution in any way. So even if the remove expression was not defined for the surpriseDiscount instance pointcut, when the Product instances are collected by the garbage collector, they are removed from the set.

4.2 Multisets

An instance pointcut reifies an object set as a *multiset*. A multiset, also referred to as a *bag*, allows multiple appearances of an object. Every contained object has a corresponding cardinality which indicates its multiplicity in the set.

The instance pointcut shown in Listing 4 selects Product instances, which are applied a Discount. The remove expression removes a Product instance if the Discount is removed from that Product. With this pointcut we would like to represent the currently discounted products. Multiset makes sure that Products can be added for each discount submission operation. When the same product is added with different types of discounts, and if one of the discounts is removed, then still one entry of that instance is left in the set. If instance pointcuts only supported a set then as soon as a discount is removed from a product, its only copy would be removed and it would appear as if there are no more discounts on that product.

4.3 Refinement and Composition

Instance pointcuts can be referenced by other instance pointcuts. They can be refined in two ways and they can be composed together to create new instance pointcuts.

```
1 static instance pointcut multi_discount<Product>:
2   after(call(* ProductManager.submitDiscount(..))
3   && args(instance))
4   UNTIL
5   after(call(* ProductManager.withdrawDiscount(..))
6   && args(instance));
```

Listing 4: An instance pointcut utilizing multiset property

4.3.1 Referencing and Type Refinement

Instance pointcuts are referenced by their names. Optionally the reference can also take an additional statement for *type refinement*, which selects a subset of the instance pointcut that is of the specified type. Type refinements require that the refinement type is a subtype of the original instance type. For example the instance pointcut surpriseDiscount (Listing 3) can be refined as shown in Listing 5. The refinement expression selects the subset of BeautyProduct instances from the set of Product instances selected by the surpriseDiscount instance pointcut. The surpriseDiscountBeauty instance pointcut is defined using the result of this expression. Note that with this notation objects that are of a subtype of BeautyProduct will also be selected.

```
1 static instance pointcut ↙
      surpriseDiscountBeauty<BeautyProduct>:
2   surpriseDiscount<BeautyProduct>;
```

Listing 5: A type refined pointcut

4.3.2 Instance Pointcut Expression Refinement

In section 4.1 we have introduced the instance pointcut expression, which consists of two sub-expressions (add and remove expressions). We provide an expression refinement mechanism which makes it possible to reuse parts of the existing instance pointcut expressions to create new ones. The expression elements forming the sub-expressions can be accessed individually to be extended by concatenating other primitive pointcuts, so-called *refinement expressions*, with boolean operators. We offer a naming convention to access parts of the instance pointcut expression with different granularity. Note that this syntax is only valid when used in the context of an expression refinement.

¡ip-ref¿ When an instance pointcut is referenced directly then the refinement expression is composed with the pointcut expression in all of the before and after event selectors, in the add and remove expressions.

¡ip-ref¿.add{**remove**} This expression provides access at the sub-expression level. The refinement expression is composed with the pointcut expressions in referenced sub-expression's before and after event selectors.

¡ip-ref¿.add{**remove**}_**after**{**before**} This naming convention is used to access the pointcut expressions of the individual before and after event selectors and provides the finest granularity. In fact the other two access statements can be written in terms of this one, they just provide a short hand for the collective expression refinements.

It is possible to compose any primitive pointcut, except the binding predicates, with a sub-expression. Although we chose not to restrict this aspect, some compositions will not be meaningful for selecting objects. For example composing an **execution** pointcut with an expression that already includes a **call** pointcut will result in a non-matching pointcut expression. This is further discussed in Section 6.

Let us explain the usage of the expression access by examples. The example shown in Listing 6 shows a reuse of the surpriseDiscount's sub-expressions to create a new instance pointcut. The newly created pointcut's sub-expression can be accessed through the aforementioned naming conventions. In order to assign the sub-expressions we use the '**:**' assignment operator, to provide a uniform syntax. It is not possible to assign a higher granularity statement to a lower granularity one; i.e. the following is illegal add_before : ↙ <ip−ref>.add.

```
1 static instance pointcut surpriseDiscountOver50<Product>:
2   add: surpriseDiscount.add && if(instance.getPrice() > ↙
        50) UNTIL
3     remove: surpriseDiscount.remove;
```

Listing 6: Expression refinement of surpriseDiscount(Listing 3) instance pointcut

The **if** pointcut in listing 6 is appended to the add expression of surpriseDiscount (Listing 3). The effect of this composition is as follows; the **if** pointcut will be appended to all of the pointcut expressions contained in the **after** and **before** event selectors. Since the surpriseDiscount pointcut only has one after event in its add expression, the resulting add expression is equivalent to:

```
after(call(* ProductManager.submitDiscount(..)) &&
args(instance, SurpriseDiscount) && ↙
    if(instance.getPrice() > 50))
```

Expression refinements can also be used for more precise type refinements. Revisiting the example given in subsection 4.3.1, the surpriseDiscountBeauty (Listing 5) instance pointcut can be constructed to include instances with the *exact type* BeautyProduct(Listing 7). The effect is different from type refinement since surpriseDiscountOnlyBeauty does not include subtypes of BeautyProduct.

4.3.3 Instance Pointcut Composition

Instance pointcuts reify sets, for this reason we facilitate the composition in terms of set operations: *intersection and*

```
1 static instance pointcut ↙
      surpriseDiscountOnlyBeauty<BeautyProduct>:
2   surpriseDiscount && ↙
        if(instance.getClass().equals(BeautyProduct.class));
```

Listing 7: Type refinement by expression refinement

union. In Figure 3, an extended version of the grammar definition is shown. The composition of two instance pointcuts creates a *composite* instance pointcut. Different from regular instance pointcuts, composite ones are declared with the keyword **composite** and they do not have instance pointcut expressions. Instead they monitor the component instance pointcuts' set change operations and update their own set accordingly. In order to declare a set intersection the keyword **inter** and to declare a set union the keyword **union** is used. Throughout the text we will use the mathematical symbols for these operations, ∩ as intersection and ∪ as union. Since composite instance pointcuts do not have an instance pointcut expression they cannot be used in expression refinement. However they can be type-refined; the result of the type refinement of a composite instance pointcut is also a composite instance pointcut and must be declared as such.

⟨*instance-pointcut*⟩ ::= '`composite instance pointcut`'
⟨*name*⟩ ('<' ⟨*instance-type*⟩ '>')? '`:`' ...
| ⟨*comp-expr*⟩

⟨*comp-expr*⟩ ::= ⟨*comp-expr*⟩ '`inter`' ⟨*comp-expr-t*⟩
| ⟨*comp-expr-t*⟩

⟨*comp-expr-t*⟩ ::= ⟨*comp-expr-t*⟩ '`union`' ⟨*comp-expr-f*⟩
| ⟨*comp-expr-f*⟩

⟨*comp-expr-f*⟩ ::= ⟨*ip-ref*⟩
| '(' ⟨*comp-expr*⟩ ')'

⟨*ip-ref*⟩ ::= ⟨*name*⟩
| ⟨*name*⟩('<' ⟨*refined-instance-type*⟩ '>')?

Figure 3: Syntax for instance pointcut composition

The type of a composite instance pointcut must be assignment compatible to the types of the component instance pointcuts. It is also possible to leave out the type declaration and let the compiler assign the type. For a composition of two instance pointcuts, the type of the composite one can be determined depending on the relation of the types of the component instance pointcuts. For illustration of this type inference, consider the type hierarchy in Figure 4a: R is the root of the hierarchy with the direct children A and B (i.e., these types are siblings); C is a child of B. Table 4b shows four distinct cases: Either the type of one of the instance pointcuts is a super type of the other one's type (second row), or both types are unrelated (third row); and the composition can either be ∩ (third column) or ∪ (fourth column).

When composing two instance pointcuts with types from the same hierarchy, the type of the composition is the more specific type (C in the example) for an ∩ composition and the more general type (B) for an ∪ composition. When composing two instance pointcuts with sibling types, for the ∩ operation the resulting composition cannot select any types since the types A and B cannot have a common instance. The ∪ operation will again select a mix of instances of type A and B, thus composed instance pointcut must have the common super type, R in the example.

Because instance pointcuts are reified as multisets, these operations are different from the regular set operations. The definition of the intersection and union operations for multiset is given in the next definition.

DEFINITION 1. *Assume* (X, f) *and* (Y, g) *are multisets, where* X, Y *represents the elements and* f, g *represents a function which maps each element to a cardinal number.*

*The **intersection** of these sets is defined as* (V, h) *where,*

$$V = X \cap Y$$

and $\forall v \in V$ *the multiplicity of* v *is defined as*

$$h(v) = min(f(v), g(v))$$

*The **union** of these sets is defined as* (Z, i) *where,*

$$Z = X \cup Y$$

and $\forall z \in Z$ *the multiplicity of* z *is defined as*

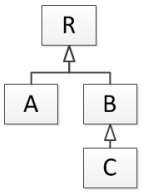
$$i(z) = max(f(z), g(z))$$

4.4 Using Instance Pointcuts

Up to now we have explained the syntax and semantics for definitions of instance pointcuts. In this section we will explain how to use instance pointcut in the context of an AO language, namely, AspectJ. As example, throughout this section, we will use the instance pointcut defined in Listing 8, which maintains a set of Products that are currently out of stock. Instance pointcuts are static members of classes and can have any visibility modifier. Thus, all modules, aspects as well as classes, that can see an instance pointcut can use it in the ways described below.

```
1 static instance pointcut outOfStock<Product>:
2   after(call(* Product.outOfStock(..)) &&
3         target(instance))
4   UNTIL
5   after(call(* Vendor.stock(..))
6         && args(instance));
```

Listing 8: An instance pointcut for out of stock products



(a) A simple type hierarchy

pc1 type	pc2 type	∩	∪
B	C	C	B
B	A	∅	R

(b) Instance pointcut compositions and effect on the captured instance type

Figure 4: An example to illustrate composition's effect on types

4.4.1 Set Access

Instance pointcuts reify a phase-specific object set and this set can be accessed through a static method, which has the same name as the instance pointcut identifier. Only the *get* methods of the collection interface can be used to retrieve objects from the set. We ensure this by returning an UnmodifiableSet from the set access methods. In listing 9 the outOfStock() method (line 4) returns the set of Products that are currently out of stock. Write methods, which modify the contents of the set, are not allowed since they create data inconsistencies like adding an object which is not in the same life-cycle phase as the ones selected by the instance pointcut. The usage of write methods may result in concurrent modification exceptions.

```
1 public static double calculateDamages()
2 {
3   double damage = 0;
4   for(Product p: MyAspect.outOfStock())
5     damage = damage + p.getPrice();
6   return damage;
7 }
```

Listing 9: Calculate a damage estimate for out of stock products

4.4.2 Set Monitoring

An instance pointcut definition defines two set change events, an add event and a remove event. In order to select the join points of these events, every instance pointcut definition automatically has two implicit regular pointcuts. These implicit pointcuts have the following naming conventions, $\langle name \rangle\_instanceAdded$, $\langle name \rangle\_instanceRemoved$, where $\langle name \rangle$ is the name of the instance pointcut. In Listing 10, a before advice using the outOfStock_instanceAdded pointcut is shown. When a product is marked out of stock and it is added to the set, a notification is sent to the related Vendor indicating that the product is out of stock.

```
1 before(Product p): outOfStock_instanceAdded(p)
2 {
3   OnlineShop.notifyVendor(p.getVendor, STOCK_MSG);
4 }
```

Listing 10: Set monitoring pointcut used to notify vendors

5. Compilation of Instance Pointcuts

A goal for our compiler implementation is to support modular compilation. This means to compile an aspect with instance pointcuts that refer to instance pointcuts defined in other aspects, it must be sufficient to know their declaration (i.e., the name and type); it should not be necessary for the compiler to know the actual expression or the referenced instance pointcuts.

We have implemented the instance pointcut language using code transformation employing two tools. First, the parser of our language and the code generation templates are implemented with the EMFText[1] language workbench. For this purpose, we have defined the AspectJ grammar by using JaMoPP[2] [10] as the foundation and extended it with the grammar for instance pointcuts which was presented interspersed with the previous section.

Second, the generated code uses the ALIA4J[3] [3] framework for so-called advanced dispatching language implementations. The term advanced dispatching refers to late-binding mechanisms including, e.g., predicate dispatching and pointcut-advice mechanisms. At its core, ALIA4J contains a meta-model of advanced dispatching declarations, called *LIAM*, in which AspectJ pointcut and advice, as well as instance pointcuts can be expressed.

We use ALIA4J to realize the crosscutting behavior of our language instead of AspectJ because the way AspectJ handles binding of values and restricting their types in pointcuts would prohibit a modular compilation of instance pointcuts. While in instance pointcuts the value binding is uniformly expressed in a pointcut expression, in AspectJ binding the result value must be specified in the advice definition (via the **after returning** keyword) and all other values are bound

[1] EMFText, see http://www.emftext.org/

[2] JaMoPP: Java Model Parser and Printer, see http://jamopp.inf.tu-dresden.de

[3] The Advanced-dispatching Language Implementation Architecture for Java. See http://www.alia4j.org/alia4j/.

in pointcut expressions. Therefore, AspectJ code generated for an instance pointcut expression would have to depend on which value is bound; this means that the code generation for a derived instance pointcut would also depend on the binding predicate (an implementation detail) of the referenced one. It is not possible to work around this, using AspectJ's reflective **thisJoinPoint** keyword, as it does not expose the result value at all. Another, similar limitation is that AspectJ does not allow to narrow down the type restriction for the bound value of a referred pointcut. Thus, in order to be able to transform an instance pointcuts with type refinement to AspectJ, it is necessary to know the definitions of the referenced instance pointcut and inline its definition.

Our compiler generates different code depending on whether the instance pointcut is a composite one or not and whether it is a refinement of an instance pointcut or not. Common to all cases is the code for managing the data of the instance pointcut. Listing 11 exemplary shows that code; the variables *${Type}* and *${ipc}* stand for the instance pointcut's type and name, respectively. The natural text written in the comments provides a description of the code for which it stands.

First, to store the instances currently selected by an instance pointcut as a multiset, a WeakHashMap is defined (cf. line 1); the keys of the map are the selected objects and the mapped value is the cardinality. We use weak references to avoid keeping objects alive which are not reachable from the base application anymore. The generated method *${ipc}* returns all objects which are currently mapped (cf. lines 2–4).

Furthermore, methods are generated to access, increase or decrease the counter of selected objects; if an object does not have an associated counter yet or the counter reached zero, the object is added to or removed from the map, respectively (cf. lines 5–15). After having performed their operations, ${ipc}_addInstance and ${ipc}_removeInstance methods invoke an empty method, passing the added or removed object. We generate a public, named pointcut selecting these calls, exposing the respective events (cf. lines 18 and 19).

Next, these bookkeeping methods have to be executed at events corresponding to the instance pointcut definitions. Below, we elaborate on the code generation for instance pointcuts defined in the different possible ways.

```
1  private static WeakHashMap<${Type}, Integer> ↙
       ${ipc}_data = new WeakHashMap<${Type}, ↙
       Integer>();
2  public static Set<${Type}> ${ipc}() {
3    return Collections.unmodifiableSet(${ipc}_data.keySet());
4  }
5  public static void ${ipc}_addInstance(${Type} instance) ↙
       {
6    //increase counter associated with instance by the ↙
         ${ipc}_data map
7    ${ipc}_instanceAdded(instance);
8  }
9  public static void ${ipc}_removeInstance(${Type} ↙
       instance) {
10   //decrease counter associated with instance by the ↙
         ${ipc}_data map
11   //if the counter reaches 0, remove instance from the map
12   ${ipc}_instanceRemoved(instance);
13 }
14 public static int ${ipc}_cardinality(${Type} o) {..}
15 private static void ${ipc}_setCardinality(${Type} o, int ↙
       c){..}
16 private static void ${ipc}_instanceAdded(${Type} ↙
       instance) {}
17 private static void ${ipc}_instanceRemoved(${Type} ↙
       instance) {}
18 public pointcut ${ipc}_instanceAdded(${Type} instance) ↙
       : call(private static void ↙
       Aspect.${ipc}_instanceAdded(${Type})) && ↙
       args(instance);
19 public pointcut ${ipc}_instanceRemoved(${Type} ↙
       instance) : call(private static void ↙
       Aspect.${ipc}_instanceRemoved(${Type})) && ↙
       args(instance);
```

Listing 11: Template of generated code for instance set management.

5.1 Non-Composite Instance Pointcuts

A non-composite instance pointcut, generally consists of four underlying pointcut definitions: specifying join points (1) *before* or (2) *after* which an instance is to be *added* to the selected instances; and specifying join points (3) *before* or (4) *after* which an instance is to be *removed*. For each pointcut definition, we generate a method that creates a corresponding LIAM model; the methods are called *${ipc}*_add_before, *${ipc}*_add_after, *${ipc}*_remove_before, and *${ipc}*_remove_after.

In LIAM, a *Specialization* can represent a partial AspectJ pointcut and a full pointcut expression can be represented as the disjunction of a set of *Specializations* (discussed in detail elsewhere [2]). Figure 5 shows the meta-model for a Specialization in ALIA4J consisting of three parts. A *Pattern* specifies syntactic and lexical properties of matched join point shadows. The *Predicate* and *Atomic Predicate* entities model conditions on the dynamic state pointcut designators depend on. The *Context* entities model access to values like the called object or argument values. Contexts which are directly referred to by the Specialization are exposed to associated advice (i.e., they represent binding predicates).

Depending on the definition of the instance pointcut, LIAM models of the underlying pointcuts have to be created in different ways. All four underlying pointcuts are optional; a missing pointcut can be represented as an empty set of Specializations in LIAM.

Plain Instance Pointcuts For pointcut expressions that are directly provided, we use a library function provided by ALIA4J which takes a String containing an AspectJ pointcut

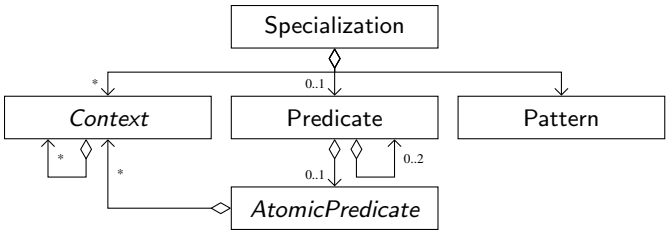


Figure 5: Meta-model of a Specialization in ALIA4J.

as input. We have extended this library to also accept the **returning** pointcut designator.

```
1 static instance pointcut ${ipc}<${Type}>:
2 before(${pc_add_before}) after(${pc_add_after}) UNTIL
3 before(${pc_remove_before}) after(${pc_remove_after};
```

Listing 12: Example of a plain instance pointcut

For the example instance pointcut presented in listing 12, we show the code generated for the method creating the LIAM model for the add_before pointcut in listing 13; the other methods are generated analogously. Line 4 shows the transformation of an AspectJ pointcut into a set of Specializations in the LIAM meta-model by passing the pointcut as a String—represented by ${pc_add_before} in listings 12 and 13—to the above mentioned library function.

```
1 private static Set<Specialization> ${ipc}_add_before;
2 public static Set<Specialization> ${ipc}_add_before() {
3   if (${ipc}_add_before == null) {
4     ${ipc}_add_before = ↙
          Util.toSpecializations("${pc_add_before}", ↙
          ${Type});
5   }
6   return ${ipc}_add_before;
7 }
```

Listing 13: Template for creating the LIAM model for the add_before expression

Type Refinement An instance pointcut can also be defined by referring to another instance pointcut whereby a type restriction for the selected instances can be defined. A template for an instance pointcut defined as such is as follows:

```
static instance pointcut ${ipc}<${Type}>: ↙
    ipc1<${Type}>;
```

For such an instance pointcut also methods are created to produce LIAM models for the four underlying pointcuts (see the template in listing 14). These methods first invoke the corresponding methods of the referenced instance pointcut (cf. line 3). Second, the type restriction is added to the Predicates of the retrieved Specializations (cf. line 4) with the method addTypeConstraint.

```
1 public static Set<Specialization> ${ipc}_add_before() {
2   ...
3   Set<Specialization> ipRef = ipc1_add_before();
4   ${ipc}_add_before = Util.addTypeConstraint(ipRef, ↙
        ${Type});
5   ...
6 }
```

Listing 14: Template for creating the LIAM model for the type-refined instance pointcut

Expression Refinement When defining a new instance pointcut through expression refinement, for each of the four underlying pointcut expressions, a plain pointcut expression can be *and*ed or *or*ed with the underlying pointcut expression of the referenced instance pointcut. As explained in the previous section, refinement pointcut expressions must not include a binding predicate. The referred instance pointcut expression already has a binding predicate which is carried over to be used as the binding predicate for the newly composed pointcut expression. The template for the generated method for creating the add/before LIAM model is shown in listing 15. It assumes that the instance pointcut is named ${ipc} and it refines another instance pointcut ${ipc1} by *and*ing the pointcut expression ${ipc1_add_before} to the add/before underlying pointcut. If the pointcuts are *or*ed, correspondingly the method orSpecializations is used in line 5.

These utility methods are provided as runtime library for the instance pointcuts. The method andSpecializations forms the conjunction of the Predicates and Patterns of the passed Specialization sets. If ipRef is empty, the conjunction is also empty and an empty set is returned. Otherwise, the Context declared by the Specializations ipRef is copied to the ones newly created by the andSpecializations method. The method orSpecializations is implemented similarly. But it forms the disjunction of Predicates and Patterns and if ipRef is empty, an exception is raised.

```
1 public static Set<Specialization> ${ipc}_add_before() {
2   ...
3   Set<Specialization> plainIPExpr = ↙
        Util.toSpecializatons("${pc_add_before}", ${Type});
4   Set<Specialization> ipRef = ipc1_add_before();
5   ${ipc}_add_before = Util.andSpecializations(ipRef, ↙
        plainIPExpr);
6   ...
7 }
```

Listing 15: Generated code for creating the LIAM model for the add/before pointcut of the instance pointcut created with expression refinement

Deployment In each of the above cases, the created LIAM models of the pointcuts must be associated with advice invoking the add or remove method for the instance pointcut. In a LIAM model this is achieved by defining an Attachment, which roughly corresponds to a pointcut-advice pair. An Attachment refers to a set of Specializations, to an *Action*, which specifies the advice functionality, and to a *Schedule Information*, which models the time relative to a join

```
1  public static void ${ipc}_deploy() {
2    org.alia4j.fial.System.deploy(
3      new Attachment(
4        ${ipc}_add_before(),
5        createStaticAction(void.class, ${Aspect}.class, ↙
             "${ipc}_addInstance", new ↙
             Class[]{${Type}.class})
6        SystemScheduleInfo.BEFORE_FARTHEST),
7      //Create Attachments for the other three parts ↙
           analogously.
8      //For the ''after'' parts, use ↙
           SystemScheduleInfo.AFTER_FARTHEST.
9      //For the ''remove'' parts, specify method ↙
           ${ipc}_removeInstance.
10     );
11 }
```

Listing 16: Deployment of the bookkeeping for an instance pointcut.

```
1  public static void ${ipc}_update(Object o) {
2    int oldCardinality = ${ipc}_cardinality(o);
3    int newCardinality =
4      Math.min(
5        Math.max(
6          ${ipc1}_cardinality(o),
7          ${ipc2}_cardinality(o)),
8        ${ipc3}_cardinality(o));
9    ${ipc}setCardinality(o, newCardinality);
10   if (oldCardinality == 0 and newCardinality > 0)
11     ${ipc}_instanceAdded(o);
12   else if (oldCardinality > 0 and newCardinality == 0)
13     ${ipc}_instanceRemoved(o);
14 }
```

Listing 17: The update method generated from a composition expression

point when the action should be executed, e.g., "before" the join point (cf. line 6).

Listing 16 shows the generated code for creating and deploying the bookkeeping Attachments. The first Attachment uses the set of Specializations returned by the ${ipc}add_before method (cf. line 4) and specifies the ${ipc}_addInstance method as action to execute at the selected join points (cf. line 5). As relative execution time, the Attachment uses a "SystemScheduleInfo"; this is provided by ALIA4J for Attachments performing maintenance whose action should be performed before or after all user actions at a join point, such that all user actions observe the same state of the maintained data. Thus, when reaching a selected join point the instance is added to the instance pointcut's multiset before any other action can access its current content. The other Attachments are created analogously. In the end, all Attachments are deployed through the ALIA4J System (cf. line 2).

5.2 Composite Instance Pointcuts

Composite instance pointcuts require a different compilation strategy because they do not have the four underlying pointcut expressions. The data of a composite instance pointcut changes when the data of one of its referenced instance pointcuts is updated. The corresponding events happen during the execution of the generated methods ${ipc}_addInstance and ${ipc}_removeInstance. Therefore, a different mechanism is needed than for the non-composite instance pointcuts which depend on user events.

Intersection and Union When an instance pointcut ${ipc} is composed by forming the union or intersection of other instance pointcuts (${ipcX}), the content of the maintained multiset potentially changes whenever an instance is added to or removed from one of the referenced instance pointcuts—events already exposed through ${ipcX}_instanceAdded or ${ipcX}_instanceRemoved pointcuts. For the maintenance of a composite instance pointcut a method is generated which reacts to the join points matching the disjunction of all these pointcuts. The argument of this method is the instance exposed by these pointcuts, i.e., the instance that has either been added to or removed from a reference instance pointcut. When the maintenance method is invoked, we know the cardinality of this instance potentially changes in the multiset of the instance pointcut ${ipc}. The cardinality of other instances cannot change. The generated method, therefore, re-calculates the cardinality of the affected instance and changes its value in ${ipc}_data.

To generate appropriate code, the compiler first builds a binary expression tree for the composition expression. Next, it traverses this tree and generates different code for the cases that the visited node is an instance pointcut reference, or an `inter` or `union` operator. For an instance pointcut reference, code is generated that retrieves the cardinality of the instance in the multiset of the referenced instance pointcut. For an `inter` and `union` operator, code is generated that calculates the minimum and maximum, respectively, of both sub-expressions. Finally, the cardinality in ${ipc}_data is updated.

As example, listing 17 shows the generated code for a composite instance pointcut with the set expression $({ipc1} ↙ **union** ${ipc2})**inter** ${ipc3}. Besides, the generated code remembers the old cardinality; when the cardinality changes from 0 to > 0 or vice versa, the generated method invokes the method ${ipc}_instanceAdded ${ipc}_instanceRemoved, respectively.

As in the case of non-composite instance pointcuts, a LIAM Attachment is generated and deployed which associates the Specializations corresponding to the pointcuts with the generated method.

Type Refinement for Composite Instance Pointcuts Instance pointcuts which are defined by means of type refined

composite instance pointcuts, are treated similar to the case above. A method is generated which is executed when the referenced instance pointcut changes. The method checks whether the type of the added or removed object is assignment compatible with the type restriction. If this is the case, the same operation (adding or removing the instance) is performed on the multiset of the refining instance pointcut.

5.3 Compiling Plain AspectJ constructs

To ensure consistent ordering between AspectJ advice and our implementation of instance pointcuts (i.e., that our bookkeeping advice are executed before user advice), the AspectJ pointcut-advice definitions must be processed by ALIA4J. This is possible because ALIA4J can integrate with the standard AspectJ tooling. Using command line arguments the AspectJ compiler can be instructed to omit the weaving phase. The advice bodies are converted to methods and pointcut expressions are attached to them using Java's annotations which are read by the ALIA4J-AspectJ integration and transformed into Attachments at program start-up. The code generated by our compiler consists of the above explained methods, as well as plain AspectJ definitions. When compiling this code with the mentioned command line options, the regular AspectJ pointcut-advice and the behavior of the instance pointcuts are both executed by ALIA4J, thus ensuring a consistent execution order.

6. Validation

The instance pointcuts approach satisfies the goals we have stated at the beginning of Section **??** and in Section 5 we have shown that instance pointcuts can be compiled modularly. Our approach provides:

- A concise syntax, which is used to generate the necessary book-keeping code
- additional features to declaratively create refined sets, reusing already created instance pointcuts
- a composition mechanism, which uses set operations, that allows modular definition of instance pointcuts.

Let us revisit the problem we have identified in section 2 and provide an implementation that uses instance pointcuts shown in listing 18 (this listing only includes the changed code, the rest is identical to listing 2). The instance pointcut surpriseDiscount successfully encapsulates the bookkeeping concern, which reduces the concern specific lines of code to a third of the AspectJ solution. The discount alert advice now uses the surpriseDiscount_instanceAdded set monitoring pointcut (lines 4 – 6). Also, in order to access the instance pointcut set, the code now uses the set access method as in line 10. Introducing new concerns to this code, like refining the surpriseDiscount instance pointcut to select objects with prices that are higher than 50$ (Listing 6) requires very little amount of code.

```
1  aspect SDiscount{
2    static instance pointcut surpriseDiscount<Product>: ↙
         after(call(* ProductManager.submitDiscount(..)) ↙
         && args(instance, SurpriseDiscount))
3      UNTIL after(call(* ↙
           ProductManager.withdrawDiscount(..)) && ↙
           args(instance, SurpriseDiscount));
4    after(Product p):surpriseDiscount_instanceAdded(p){
5      OnlineShop.INSTANCE().createDiscountAlert(p);
6    }
7    void around(String listType):display(listType){
8    ...
9        OnlineShop.instance().getUI()
10         .display(SDiscount.surpriseDiscount());
11   }...
12 }
```

Listing 18: The instance pointcut implementation of the discount alert concern

A declarative syntax such as that of instance pointcuts, generally allows performing various checks, generate well-placed error or warning markers and informative warning/error messages. In the following we discuss checks we deem useful and possible to implement based on our language. Proving their feasibility by implementing the checks is subject to future work.

Non-existent/Incompatible types If the type declared by the instance pointcut does not exist, this is a compile error. Instance pointcuts provide an additional check during type refinement; it is a compile error if the refinement type is not a subtype of the referenced pointcut's declared type. In the Java solution the error marker is placed at the line of the instanceof check, without giving any context to why the **instanceof** check is performed. The instance pointcut error marker is placed at line of the refinement, with an elaborate error message explaining the type mistake.

Type compatibility is also an issue while composing instance pointcuts. The typing effects of the composition is previously mentioned in section 4.3.3. If in a composite instance pointcut declaration, the instance type is explicit, then the type compatibility between the composite and the component instance pointcuts' should be checked. The compiler determines the appropriate type for every composite instance pointcut, whether the instance type is explicit or not. The computed type is then compared to the declared type; the declared type must be the same or a super type of the computed type. If there's an incompatibility, we put an error marker at the line where the type was declared, indicating the composed type is not compatible with the declared type of the composite instance pointcut.

Empty Sets The add expression of an instance pointcut is responsible for populating the instance pointcut set. If the pointcut expressions defined in the add expressions do not

match any join-points then it is guaranteed that the instance pointcut set will always be empty. This case is displayed as a compile-time warning which indicates that no objects will be selected. During the expression refinement, composition of the new pointcuts and the referenced one may result in non-matching pointcut expressions. Currently we do not perform any control-flow analysis to check for empty sets. Such a check will help us to identify cases where even if the add expression matches, no objects are added to the set.

Double selection Instance pointcut syntax allows to select the same join-point in both before and after events in the same sub-expression. In Listing 19 such a case is illustrated. The Product object is added before it is applied a discount, and once again after the discount is applied. This will increment the cardinality of the same object twice. The analogous case exists for the remove expressions. The object is removed from the set *twice*. This behaviour is consistent with an instance pointcut's regular behaviour, but we still raise a separate warning for the add and remove cases to notify the developer, in case of a copy-paste error.

```
1 static instance pointcut surpriseDiscountDouble<Product>:
2   before(call(* ProductManager.submitDiscount(..))
3   && args(instance, SurpriseDiscount))
4   ||
5   after(call(* ProductManager.submitDiscount(..))
6   && args(instance, SurpriseDiscount))
```

Listing 19: Adding the same object before and after the same join-point

Expression Refinement Checks During expression refinement there is a special case when the refined instance pointcut is referencing a non-existent event selector and the || boolean operator is used. Assume that ipc1 only has an add_before expression. While refining this instance pointcut the developer mistakenly writes the following:

```
1 ipc2<T>: add_before:ipc1.add_before
2   add_after: ipc1.add_after || call(..);
```

This definition refers to the non-existent add_after expression. This is a compile error since, while the **call** pointcut is selecting joinpoints, no objects are bound. This is illegal to have in an instance pointcut expression. Note that if the operation was && there would be no compile error since, the add_after expression of ipc2 would simply be empty and ip2's add expression would only comprise of ipc1's add_before event selector.

7. Related Work

AO-extensions for improving aspect-object relationships are proposed in several studies. Sakurai et al. [15] proposed Association Aspects. This is an extension to aspect instantiation mechanisms in AspectJ to declaratively associate an aspect to a tuple of objects. In this work the type of object tuples are declared with a perobjects clause and the specific objects are selected by pointcuts. This work offers a method for defining relationships between objects. Similar to association aspects, Relationship Aspects [14] also offer a declarative mechanism to define relationships between objects, which are cross-cutting to the OO-implementation. This work focuses on managing relationships between associated objects. Bodden et al. [4] claim that the two above approaches lack generality and propose a tracematch-based approach. Although the semantics of the approaches are very similar, Bodden et al. combine features of thread safety, memory safety, per-association state and binding of primitive values or values of non-weavable classes. Our approach, also extending AO, differs from these approaches since our aim is not defining new relationships but using the existing structures as a base to group objects together for behaviour extensions. Our approach also offers additional features of composition and refinement.

The "dflow" pointcut [11] is an extension to AspectJ that can be used to reason about the values bound by pointcut expressions. Thereby it can be specified that a pointcut only matches at a join point when the origin of the specified value from the context of this join point did or did not appear in the context of another, previous join point (also specified in terms of a pointcut expression). This construct is limited to restricting the applicability of pointcut expressions rather than reifying all objects that match certain criteria, as our approach does.

Another related field is Object Query Languages (OQL) which are used to query objects in an object-oriented program (e.g., [5]). However OQLs do not support event based querying, which selects objects based on the events they participate in, as presented in our approach. It is interesting to combine instance pointcuts with OQL. For example instance pointcuts can be used as a predicate in OQL expressions, in order to select from phase-specific object sets. We will explore possible applications in future work.

Type States [6] allow to define a state chart for a type. This specifies which states an object of that type can be in and what causes state transitions. Similar to our approach, state transitions are triggered by runtime events. However unlike instance pointcuts, in type states an objects state can only change at method calls where the object is the receiver; with instance pointcuts, objects life-cycle phases can be defined more flexibly by referring to any event where the object is in the dynamic context, e.g., passed as argument or result value. The purpose of the type states approach is to facilitate more powerful invariant checking at compile-time, whereas we provide a mechanism to actually track object sets at runtime. It would be interesting to investigate possibilities for combining both approaches in the future.

8. Conclusion and Future Work

In this work we have presented instance pointcuts, a specialized pointcut mechanism for reifying phase-specific object sets. Our approach provides a declarative syntax for defining events when an object starts or ends to belong to a life-cycle phase. Instance pointcuts maintain multisets providing a count for objects which enter the same life-cycle phase more than once. Instance pointcut sets can be accessed easily and any changes to these sets can also be monitored with the help of automatically created set monitoring pointcuts. The sets can be declaratively composed, which allows reuse of existing instance pointcuts and consistency among corresponding multisets. Finally, we have presented our modular compilation approach for instance pointcuts based on AspectJ and the ALIA4J Language implementation architecture. ALIA4J provided us with the flexibility AspectJ lacked in instance pointcut composition and type refinement.

The syntax and expressiveness of instance pointcuts partially depend on the underlying AO language; this is evident especially in our usage of the AspectJ pointcut language in the specification of events. Since AspectJ's join points are "regions in time" rather than events, we had to add the "before" and "after" keywords to our add and remove expressions. Thus, compiling to a different target language with native support for events (e.g., EScala [8] or Composition Filters [1], the point-in-time join point model [13]) would influence the notation of these expressions.

We think the instance pointcut concept is very flexible and can be useful in various applications. It eliminates boilerplate code to a great extent and provides a readable syntax. We believe that the reuse and composition mechanisms offered by instance pointcuts are beneficial for software evolution since they make it easy to create tailored variations according to new requirements.

One relevant field of application for instance pointcuts are design patterns which are known to be good examples for aspect-oriented programming [9]. Many design patterns exist for defining the behaviour for groups of objects; thus implementing them with our instance pointcuts seems to provide a natural benefit. We are currently working on creating object adapters wrapping the objects reified by an instance pointcut.

References

[1] L. M. Bergmans and M. Aksit. How to deal with encapsulation in aspect-orientation. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[2] C. Bockisch and M. Mezini. A flexible architecture for pointcut-advice language implementations. In *Proceedings of the 1st Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms, VMIL*. ACM, 2007.

[3] C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at ALIA4J. *Journal of Object Technology*, 11(1):1–28, 2012.

[4] E. Bodden, R. Shaikh, and L. Hendren. Relational aspects as tracematches. In *Proceedings of the 7th international conference on Aspect-oriented software development*, pages 84–95. ACM, 2008.

[5] S. Cluet. Designing oql: Allowing objects to be queried. *Information systems*, 23(5):279–305, 1998.

[6] R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, editor, *ECOOP 2004 - Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer Berlin Heidelberg, 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[8] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. Escala: modular event-driven object interactions in scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240, New York, NY, USA, March 2011. ACM.

[9] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[10] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the gap between modelling and java. In M. van den Brand, D. Gaševic, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 374–383. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-12107-4_25.

[11] K. Kawauchi and H. Masuhara. Dataflow Pointcut for Integrity Concerns. In B. de Win, V. Shah, W. Joosen, and R. Bodkin, editors, *AOSDSEC: AOSD Technology for Application-Level Security*, Mar. 2004.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. *ECOOP 2001 Object-Oriented Programming*, pages 327–354, 2001.

[13] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. In N. Kobayashi, editor, *Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147. Springer Berlin / Heidelberg, 2006. 10.1007/11924661_8.

[14] D. Pearce and J. Noble. Relationship aspects. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 75–86. ACM, 2006.

[15] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matuura, and S. Komiya. Design and implementation of an aspect instantiation mechanism. *Transactions on aspect-oriented software development I*, pages 259–292, 2006.