Expressive Scoping of Dynamically-Deployed Aspects

Éric Tanter *

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
etanter@dcc.uchile.cl

Abstract

Several aspect languages and frameworks have recognized the need for dynamic deployment of aspects. However, they do not provide sufficiently expressive means to precisely specify the scope of deployed aspects. As a result, programmers have to resort to unnecessarily complex pointcut definitions that hinder the reuse potential of aspects. To address the issue of precise and expressive scoping of aspects at deployment time, we propose *deployment strategies* for parameterized dynamic aspect deployment. This novel mechanism gives full control over the propagation of the aspect on the call stack and within created objects or functions, and permits a deployment-specific refinement of its pointcuts. We discuss and illustrate the gain in expressiveness, and provide the operational semantics of deployment strategies with Scheme interpreters, for both functional and object-oriented based aspect languages.

1. Introduction

In the pointcut-advice (PA) mechanism for aspect-oriented programming [25, 36], as embodied in AspectJ [21] and others, crosscutting behavior is defined by means of pointcuts and advices. Points during execution at which advices may be executed are called (dynamic) join points. A pointcut identifies a set of join points, and an advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and advices. In AspectJ, the decision of whether or not to use an aspect within a program is done at build time; if so, the aspect has global scope, *i.e.* it *sees* all join points of the program execution. Restricting the scope of an aspect can be done by introducing conditions in the pointcut definitions.

This however renders pointcut definitions unnecessarily complex and sacrifices the reuse potential of aspects [2, 13, 28]. Also, the exact dynamic patterns under which an aspect should be effective may be impossible to foresee or very hard to express in the aspect definition. Furthermore, when analyzing the potential problems that can arise when importing modules containing woven aspects, McEachen and Alexander also make clear that developers need more control over scoping of aspects [26].

* Partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and FONDECYT project 11060493.

AOSD'08, March 31 – April 4, 2008, Brussels, Belgium.
Copyright © 2008 ACM 978-1-60558-044-9/08/0003…$5.00

Dynamic deployment permits designers to address the tradeoff between specificity and reusability by deferring certain decisions to aspect deployment, leaving aspect definitions more reusable. Using dynamic deployment, the programmer has explicit control over the scope of an aspect. Dynamic aspect deployment has also been shown to better support software variability [29, 31].

A number of aspect languages and frameworks hence support dynamic aspect deployment, under different flavors and scoping semantics. For instance, CaesarJ [2] supports per-thread aspect deployment, `deploy(a)`{*block*}, whereby the aspect instance `a` sees all join points produced in the dynamic extent of the execution of *block*. This mechanism is also found *e.g.* in AspectScheme [16] and AspectS [20]. There also exist mechanisms to deploy aspects on particular objects [2, 31], or globally [2, 19, 31, 33]. AspectScheme supports lexical scoping too, whereby an aspect sees all join points that are lexically visible in *block*, including those occurring within nested function definitions, even if the function is used outside of the block [16]. Dynamic and lexical scoping in AspectScheme are introduced in Sect. 2.

All the above deployment mechanisms are limited in that their scoping semantics only represent a fixed set of solutions in the design space. For instance, when deploying an aspect over the dynamic extent of a block, one may need to specify certain *bounds* to this extent, beyond which the aspect does not propagate anymore; *e.g.* to avoid seeing join points occurring behind the access to a particular facade object in the system. As another example, consider that when deploying a monitoring aspect, one may want to avoid monitoring the activity of certain dynamically-determined sensitive objects. The problem is that existing mechanisms only permit to deploy an aspect instance *as is*: it is not possible to specify a deployment-specific refinement, that makes it possible to *filter out* particular join points *within* the extent of the aspect deployment. These kinds of strategies should possibly be specified at deployment time, and not require a modification of the aspect definition itself. We further illustrate these limitations and others in Sect. 3.

In order to address the issue of precise and expressive scoping of aspects at deployment time, we propose a novel mechanism called deployment strategies, which gives full control over *(a)* the propagation of the aspect on the call stack, *(b)* its propagation within created objects or functions, and *(c)* deployment-specific refinement of its pointcuts. Sect. 4 gives an informal presentation of our proposal, and shows how deployment strategies concisely express the scenarios considered in Sect. 3. To set the base for the operational semantics of deployment strategies, Sect. 5 describes the semantics of scoping in AspectScheme. We extend this description for deployment strategies (Sect. 6) and present Scheme interpreters for them (Sect. 7) in both functional and object-oriented based aspect languages. Sect. 8 discusses related work and Sect. 9 concludes.

2. Background: Static and Dynamic Aspects

Dutchyn, Tucker and Krishnamurthi studied the introduction of pointcut-advice AOP to a language with higher-order functions like Scheme [16]. The proposed language, AspectScheme, extends Scheme with global top-level aspects (not discussed here), as well as statically- and dynamically-scoped aspects, revised hereafter. We also discuss the transposition of these scoping mechanisms to objects, and their applications.

2.1 AspectScheme in a Nutshell

In AspectScheme, join points are function applications (for simplicity, no discrimination is made between function application and execution [16]). A join point can be either top-level or nested within other active function applications. So a join point in context is actually represented by a recursive structure, whose head is the function to be applied, and whose tail is the active functions. Both pointcuts and advices are first-class values. A pointcut is a predicate over join points in context: it is a function of type $\mathcal{PC} = JoinPoint \rightarrow Bool$[1]. A `call` pointcut designator (*i.e.* a function that returns a pointcut) can be defined as follows:

```
(define call
  (lambda (f) (lambda (jp) (eq? f (car jp)))))[2]
```

If `open-file` is a function in scope, `(call open-file)` returns a pointcut that matches application of that function. An advice is a join point transformer, that takes a join point as parameter and returns a new function to use instead. It is a function of type $\mathcal{ADV} = JoinPoint \rightarrow Val \rightarrow Val$, *e.g.*:

```
(define trace
  (lambda (jp) (lambda (arg) (printf "calling")
                             (app/prim jp arg))))
```

The `trace` advice, for a join point `jp`, returns a function that when applied to the argument, emits a trace and performs the original function application. `app/prim` is similar to `proceed` in AspectJ, and ensures that applying `jp` will not invoke any further aspects.

2.2 Scoping Strategies

As pointcuts and advices are first-class values, AspectScheme introduces an expression to dynamically deploy an aspect over a body expression. Thus scoping considerations appear: it is necessary to define the precise *extent* of the jurisdiction of the aspect, *i.e.* what join points it will see.

The authors build upon the familiar reasoning of scope for variables: in a first-class function, statically-scoped variables get their values from the environment in which the function was *defined*; in constrast, dynamically-scoped variables get their values from the environment of the function *application*.

Consequently, two aspect deployment expressions are introduced (both of type $\mathcal{PC} \rightarrow \mathcal{ADV} \rightarrow Exp \rightarrow Val$):

- `fluid-around` deploys a dynamically-scoped aspect. Such an aspect sees all join points occurring in the dynamic extent of its body. This is essentially the same mechanism as a `deploy` expression in CaesarJ.
- `around` deploys a statically-scoped aspect. Such an aspect only sees join points occurring lexically in its body, *including* those of unapplied functions, which are exported from the body.

[1] Formalizing pointcuts as functions of type $JoinPoint \rightarrow Bool$ does not take into account the fact that generally pointcuts –and in this case, Scheme functions– can access mutable state that we ought to model explicitly. However this would only obscure the main points we are focusing on.

[2] The `eq?` primitive of AspectScheme compares functions by considering both the source locations and the captured lexical environments [16].

```
(let ((apply-to-brussels (lambda (f) (f "brussels"))))
  (fluid-around (call open-file) trace
    (apply-to-brussels open-file)))
```

Above, the `let` defines and binds a higher-order function that takes another function `f` as parameter and applies it to `"brussels"`; then, we dynamically deploy an aspect consisting of the `trace` advice defined previously and a call pointcut. Then we apply the function `apply-to-brussels` to `open-file`. When `open-file` is finally applied (the framebox above), the trace aspect *does see* the join point, because the application occurs in the dynamic extent of the `fluid-around` body expression. Conversely, in:

```
(let ((traced-open (fluid-around (call open-file) trace
                      (lambda (f) (open-file f)))))
  (traced-open "brussels"))
```

The aspect does *not* apply, because the dynamic extent of the aspect body only consists of a function *definition*. The (framed) application of `open-file` is only in the *lexical* scope of the `fluid-around` body expression: later applications are out of reach. Statically-scoped aspects serve exactly this purpose:

```
(let ((traced-open (around (call open-file) trace
                      (lambda (f) (open-file f)))))
  (traced-open "brussels"))
```

The aspect applies, because the application of `open-file` occurs *lexically* in the body of `around`. As a consequence, the aspect is "engrained" in the function that is exported from the `around` expression and bound to `traced-open`. At future applications of the function, the aspect will see the corresponding join point.

2.3 Static Aspects for Objects

Dynamically-scoped aspects have attracted considerable attention in the object-based aspect community. Examples are their incarnation in the CaesarJ language, as well as in related mechanisms outside of the pointcut-advice family, like dynamic mixin layers as provided by ContextL [11].

In contrast, statically-scoped aspects for objects have received little attention. Indeed, just like a first-class function, an object is a computational entity that embeds code (methods) to be evaluated at a later time. The notion of statically-scoped aspects in AspectScheme can therefore be transposed to objects: as a first approximation in a classless object-oriented language like Self [35], if an object is created in the lexical scope of the deployment block of an aspect, the aspect is engrained in the object. To the best of our knowledge, this has however not been considered; the closest proposals explore per-object aspect deployment (*a.k.a.* instance-local advising) [2, 31], which is quite different semantically, as will be shown later when comparing existing scoping models (Sect. 8). Also, Warth *et al.* proposed statically-scoped adaptations for objects [37], but this mechanism addresses the issue of augmenting the structure of a class, much like the inter-type declarations mechanism of AspectJ. Instead, this paper focuses on the behavioral part of adaptation with the pointcut-advice mechanism.

Because many aspect languages are based on object orientation, we are interested in bringing the analysis of scoping to both the functional and object-oriented worlds. We therefore use object-oriented scenarios to illustrate both the current state of affairs, its limitations, and the applications of our proposal.

2.4 Applications

Both static and dynamic aspects have their utility; the two scoping semantics complement each other. Dynamic aspects give control on the computation that occurs within a given dynamic extent, for instance to trap certain calls in an untrusted program [16]. The

```
class CarFactory {
  Car get(...){
    if(...) { return new Lotus(); }
    else if(...) { return new Jaguar(); }
    ...
} }
Car c = CarFactory.get(...);
SpeedLimit sl =
  speedLimit(...) ? new SpeedLimit(120) : null;
deploy-d(sl){ c.drive(); }
```

Figure 1. Usage-based variability with `deploy-d`.

```
class CarFactory {
  Car get(...){
    SpeedLimit sl =
      speedLimit(...) ? new SpeedLimit(120) : null;
    deploy-s(sl){
      if(...) { return new Lotus(); }
      else if(...) { return new Jaguar(); }
      ...
} } }
Car c = CarFactory.get(...);
c.drive();
```

Figure 2. Production-based variability with `deploy-s`.

interest of dynamically-scoped aspects has also been shown for context-dependent adaptation [11], as well as variability management [29]: the variant of an aspect that must be applied for a given usage scenario can be determined dynamically. We call this *usage-based* variability (Fig. 1): the *client* of the `CarFactory` can choose to deploy or not the `SpeedLimit` aspect (with `deploy-d`), in order to ensure that the car cannot be driven too fast.

Conversely, static aspects allow the encapsulation of crosscutting features of library functions, so that the exported functions use the aspects whenever applied. Dutchyn *et al.* use this mechanism to engrain permission checking aspects within functions of a simple operating system API that can be used by untrusted clients [16]. In terms of variability management, we call this *production-based* variability. On Fig. 2 it is now the factory that takes the decision of whether or not `SpeedLimit` is to be added to the created car. The aspect is deployed as statically scoped (with `deploy-s`), therefore it gets engrained in the created car, similarly to what happens for function definitions in AspectScheme. The client is unaware of this feature, and simply runs `c.drive()`.

3. The Need for More Expressive Scoping

The two options of the scoping model above represent a notable gain in expressiveness compared to statically-deployed aspects. However, in the following we present several dynamic deployment scenarios for which neither `deploy-d` nor `deploy-s` are sufficient. We will therefore enumerate and informally describe a number of desired operators `deploy-1`, `deploy-2`, ..., which solve these scenarios. Sect. 4 then overviews our proposal and expresses these desired operators concisely.

3.1 Case 1: Simple Refactoring

Consider the following example in AspectScheme, which applies a static billing aspect to a service function:

```
(define (get-service)
  (around billing-pc billing-adv
    (lambda ...))) ;; the actual service definition
```

```
abstract aspect QAReport {
  abstract pointcut abnormalEvent();
  before() : abnormalEvent(){
    QASite.notify(thisJoinPoint);
} }
class CarFactory {
  Bill order(...){
    QAReport qa = (...)? new QALevel1() : new QALevel0();
    deploy-2(qa){
      ProdLine.createAndDispatch(...); }
    ...
} }
```

Figure 3. Abstract and reusable QA aspect, and its dynamic deployment.

The returned service function embeds the billing aspect so that its usages are affected. But now, consider a simple refactoring:

```
(define (get-service)
  (around billing-pc billing-adv
    (get-basic-service))) ;; returns the service
```

By simply moving the definition of the function in an auxiliary function, the billing aspect is not captured anymore within the service function, because the definition of that function is now *lexically outside* of the aspect body. Deploying billing as a dynamic aspect does not solve the problem because even if the aspect sees the definition of the service function, it is not engrained in that function. This refactoring issue also manifests itself in the example of Fig. 2. Consider a refactoring whereby the cars are not instantiated directly within the lexical scope of the `deploy-s` block, but are obtained following a factory pattern, *e.g.* by calling `LotusFactory.get`: `SpeedLimit` is no longer captured at car creation time. We need a dedicated `deploy-1` expression in order to obtain a refactoring-compliant version of `deploy-s`.

3.2 Case 2: More Pervasive Propagation

Consider a quality assurance (QA) aspect in the car factory. There is an abstract, reusable, `QAReport` aspect that, upon occurrences of abnormal events, notifies the `QASite` of the factory (Fig. 3). There are different QA levels, which have different definitions of what an abnormal event is. The definition of the different QA levels is a variability that is well captured by sub-aspects in a static manner. However, the precise points at which these aspects must be deployed depends on dynamic conditions and are therefore best addressed with dynamic deployment.

The car factory must determine the QA level corresponding to the requested car, and deploy it over the call to the production line. Note that the `order` method of the car factory does not obtain a reference to the created car; it just dispatches the production, and then returns the bill to the client. Because abnormal events can be events occurring within the factory, or within the car, even after the car has been sold and dispatched to the client, the `deploy-2` mechanism must ensure that the QA aspect sees *(a)* all join points occurring within the dynamic extent of the production of the car, and *(b)* all join points occurring during the activity of cars and their elements. Therefore, `deploy-2` must be some mechanism that combines both `deploy-d` and `deploy-s`.

3.3 Case 3: More Precise Control on Propagation

Deploying `QAReport` with `deploy-2` is extreme in the sense that every single join point that derives from the call to the production line will be seen, be it produced in the dynamic extent of the production, or in the dynamic extent of the activity of any object

```
aspect SecurityPack {
  after(ProdStep p, Car c): this(p) && args(c) &&
    execution(* ProdStep.addAirBags(Car)) {
    p.addExtraAirBags(c);
  }
  around(ProdStep p, Car c): this(p) && args(c) &&
    execution(* ProdStep.setSeatBelt(Car,SeatBelt)) {
    proceed(c, new FivePointSeatBelt());
} }

class SPSetup extends ProductStep {
 SecurityPack sp = new SecurityPack();
 process(List<Car> batch) {
  List<Car> cars_sp = filterCarsWithSP(batch);
  deploy-4(cars_sp)(sp){
    super.process(batch); // triggers next step
}}}
```

Figure 4. SecurityPack aspect customizing the configuration of a car, and its dynamic deployment.

thereby created, and so on. It is highly desirable then to introduce some bounds to the pervasive propagation of Case 2.

For instance, the production line repeatedly accesses the store database through a `DBAccess` facade object; we suppose this part to be trusted and therefore there is no need to apply the QA report aspect beyond that point. We should also be able to specify that the QA aspect is only captured by `Product` objects (`Car`, `Engine`, etc.), and not by the myriad of objects that are created and used during the production process but not relevant for the purpose.

We cannot expect or require the developer of the aspect to be aware these details and prevent the propagation of the aspect in the definition itself. This would definitely sacrifice the reuse potential of the QA aspect by making it specific to a very particular case. Rather, we should use a *parametrized* `deploy-3` expression, instead of `deploy-2`, in order to obtain this more precise control over the propagation of the aspect.

3.4 Case 4: Deployment-Specific Filtering

Consider an optional security pack that can be applied to some cars, implemented as an aspect (Fig. 4). The security pack consists of adding extra airbags to the car, as well as replacing normal seat belts with 5-point belts. The aspect intervenes in the car building process, which is modeled as a pipeline of production steps: each `ProdStep` receives a batch of cars, performs its task on them and passes the batch on to the next step, invoking `super.process` (the wiring between steps is defined in the base class).

Fig. 4 shows the new production step introduced to deploy the security pack only for the cars that need it. `deploy-4` must ensure that the aspect sees the join points corresponding to the execution of the relevant production steps. This is exactly like `deploy-d`. However, the security pack aspect should only see join points that are related to those cars in the batch for which the security pack is required. This means that it should be possible to have the aspect apply only when specific, dynamically-determined instances are involved in a join point. Therefore, `deploy-4` must also be a parameterizable deployment expression.

4. Expressive Scoping of Aspects

The previous examples clearly outline that the two scoping mechanisms present in AspectScheme are insufficient to express relevant scenarios of dynamic deployment. After a brief analysis of the problem, we give a rapid overview of our proposal for expressive scoping of aspects, and show how the previous cases are succinctly expressed. Detailed description of the semantics of our proposal is deferred to the following sections.

4.1 Analysis of the Problem

The requirements of the deployment scenarios of Sect. 3 point to the fact that there are three orthogonal dimensions to be considered when deploying an aspect:

1. *Call stack propagation:* when should a deployed aspect see the join points produced beyond the activation of a new stack frame (function application, method call)?
2. *Delayed evaluation propagation:* when should a deployed aspect be captured in created procedural values (functions, objects) in order to see the join points of their future evaluations?
3. *Local join point filtering:* should a deployed aspect see *all* the join points as specified by its definition and the above propagation properties? or should all the pointcuts of the deployed aspect possibly be refined in a *deployment-local* manner, leaving the aspect definition unchanged?

The problem of existing scoping models for dynamic aspect deployment is that *(a)* they lack fine-grained control over dimensions 1 and 2, *(b)* they do not make it possible to express custom combinations of them, and *(c)* they do not consider the third dimension. As will be shown later, these three dimensions however naturally map to determining steps in the life-cycle of a deployed aspect in the interpreter.

4.2 Deployment Strategies in a Nutshell

We introduce the notion of *deployment strategies*. A deployment strategy $\delta\langle c, d, f\rangle$ specifies the scoping semantics of a dynamically-deployed aspect a with three components, corresponding to the three dimensions identified above:

- c and d are called *propagation functions*: c is used to specify whether a propagates along the call stack and d specifies propagation within delayed evaluation.
- f is called a *join point filter*. It is used to express deployment-specific filtering of the join points seen by a.

All three components are pointcuts: boolean-returning functions that take a join point in context as parameter (*i.e.* elements of $\mathcal{PC}$).

Syntax. Instead of a certain number of deployment expressions with hardwired semantics, we support one deployment expression parameterized with a deployment strategy:

$$depl(a, \delta\langle c, d, f\rangle, e)$$

depl deploys aspect a (or a list of aspects) on expression e[3] using the deployment strategy δ. When it comes to illustrating our model with a Java-like language, we use the following syntax, an extension of the CaesarJ `deploy` syntax:

```
deploy ::= deploy[c,d,f](asp){ expr }
```

As in CaesarJ, *asp* is simply an object that may contain pointcuts and advices. These pointcuts (and associated advices) are only activated when the instance is deployed. Deploying an object that has no pointcuts and advices has no effect. In each variant of the syntax (including the formal one), we use the special component '_' whenever a particular deployment strategy component is left unspecified.

[3] For uniformity, and in line with Scheme, we do not make an explicit distinction between an expression, a statement, or a block thereof.

4.3 Expressing the Examples

Cases 1 and 2. Case 1 requires the possibility to deploy an aspect that is captured in objects created in the dynamic extent of the deployment expression. This means that the call stack propagation function c must always return true. In addition, the aspect needs to be captured in created objects, therefore d must also always return true. The case is similar for `deploy-2`. So, assuming that `true` is a constant function, `deploy-2` is expressed as:

```
deploy[true,true,_](qa){
  ProdLine.createAndDispatch(...);
}
```

With this deployment strategy $\delta\langle true, true, \_\rangle$, the QA aspect will see all join points occurring within the dynamic extent of the production of the car, and will also get captured by the created car and its elements. Therefore, any abnormal events either during the production process or later, during the usage of the car, will be reported to the `QASite`.

Case 3. We can use c and d to restrict the pervasive propagation above. First, we use c to define a more precise call stack propagation strategy, whereby the propagation of the aspect on the call stack stops when the target of the call is the database facade. Hence `deploy-3` is expressed as:

```
deploy[!target(DBAccess),target(Product),_](qa){
  ProdLine.createAndDispatch(...);
}
```

Similarly to AspectJ pointcuts, `target(DBAccess)` matches when the target of the join point is of type `DBAccess`. Above we also specified the component d in a similar manner, so that the QA aspect is captured only if the created object is of abstract type `Product`.

Case 4. While the above cases dealt with propagation issues, Case 4 illustrates the need for deployment-specific tailoring of a reusable aspect, using dynamic values. `deploy-4` is expressed as:

```
deploy[true,_,if(cars_sp.contains(jp.args(1)))](sp){
  next.process(batch);
}
```

The aspect is deployed so as to propagate on the stack (c is the $true$ constant function), in order to give it dynamic scope. A join point filter is used to ensure that the security pack aspect only sees join points where the argument (denoted above with `jp.args(1)`) is a car for which the security pack applies (that is, a car contained in the `car_sp` list).

4.4 Supporting Deployment Strategies

The remainder of this paper enters in the details of the semantics of our model. In particular, we explain how to interpret deployment strategies using concise Scheme interpreters *a la* Aspect SandBox, for both functional and object-oriented base languages.

Therefore, this paper focuses on semantics, and does not address the challenges of efficient implementation in production-quality environments. Nevertheless, there exist several techniques to support dynamic deployment of aspects, at different levels: residues [18, 25], metalevel wrappers [19], optimized compilers with static analysis [3, 10], and VM support [9]. The flexibility introduced by our model represents quite a challenge for efficient dynamic deployment. However, very promising recent work on both aspect-aware VMs [7, 8] and dynamic layer (de)activation [12] suggests that such advanced scoping mechanisms can be efficiently supported.

5. Semantics of Scoping in AspectScheme

We now dive into the semantics of scoping of aspects in AspectScheme in order to set the base for the semantics of deployment strategies.

5.1 Prelude

The semantics of AspectScheme is presented in [16], using a variation on the CEKS machine [17]. For the sake of a clearer focus on scoping and a smoother transition to the object-oriented world, we reformulate these semantics as a Scheme interpreter, in the line of the Aspect SandBox (ASB) interpreters [15, 24, 25, 34].

For the sake of simplicity, and without loss of generality, we restrict ourselves to before advice. The focus of this work is on scoping, that is, how to delimit the set of join points that an aspect can potentially match; the kinds of effects at these join points is an orthogonal concern. Therefore, in contrast with the original AspectScheme description, advices are not modeled as function transformers, but simply as functions that perform their effect before the standard interpretation proceeds (their return values are ignored). We also do not account for context exposure beyond the fact that an advice receives the matched join point as parameter.

All the interpreters we discuss in this paper[4] follow the same general structure: the main function, `eval`, evaluates an expression following a simple case-based test on its type. An expression is a parsed abstract syntax tree, which can be tested with predicates like `const?`, and accessed with accessors such as `const-value`.

Compared to traditional functional interpreters, these interpreters have two distinctive features : *(a)* in order to be able to model aspect scoping precisely, an interpreter evaluates an expression within an *aspect environment* that is passed around between evaluation steps[5], in addition to the lexical environment; *(b)* in order to model join points in context, an interpreter takes as parameter the join point at the enclosing function application.

5.2 AspectScheme Semantics

In AspectScheme, an aspect environment is defined as follows:

$$A = \{\langle s, pc, adv\rangle \mid s \in \mathcal{S},\ pc \in \mathcal{PC},\ adv \in \mathcal{ADV}\}$$

An aspect in an environment, *i.e.* a *deployed aspect*, is represented as a triple consisting of the scope s, element of $\mathcal{S} = \{\texttt{static}, \texttt{dynamic}\}$, the pointcut function pc and the advice function adv. $\mathcal{PC}$ and $\mathcal{ADV}$ are as defined in Sect. 2.1.

Fig. 5 shows a simplified AspectScheme interpreter. A deployed aspect `dasp` is a simple structure (created with `make-dasp`) that has three fields, accessed with `dasp-scope`, `dasp-pc`, `dasp-adv`.

When evaluating an aspect deployment expression, the interpreter extends the current aspect environment with the deployed aspect `dasp`, and evaluates the body (eg. $\boxed{2}$). In the case of an `around` expression, corresponding to the deployment of a statically-scoped aspect, the aspect is marked with the `static` attribute $\boxed{1}$; if the expression is a `fluid-around`, the aspect scope is set to `dynamic` $\boxed{3}$.

When evaluating a function definition, the interpreter returns a closure, closing over the current lexical environment. In addition, to implement the semantics of statically-scoped aspects, a closure also closes over an aspect environment A_{def} that consists of all aspects with static scope in the current aspect environment $\boxed{4}$.

$$A_{def} = \{\langle s, pc, adv\rangle \in A \mid s = \texttt{static}\}$$

[4] The executable Scheme interpreters, along with examples, are available at: `http://pleiad.dcc.uchile.cl/research/scope`

[5] The ASB interpreter of [25] and the formal model of [36] use a global aspect environment, which is insufficient for modeling scoping *a la* AspectScheme [16] (and by extension, deployment strategies as well).

```
;; evaluate expression exp in lexical environment E and aspect environment A, with current join point jp
(define (eval exp E A jp)
  (cond ((const? exp) (const-value exp))
        ((var? exp)   (lookup (var-name exp) E))
        ((arnd? exp)  (let ((dasp (make-dasp 'static (eval (arnd-pc exp) E A jp)     ⟵ [1]
                                              (eval (arnd-adv exp) E A jp))))
                        (eval (arnd-body exp)  E (cons dasp A) jp)))     ⟵ [2]
        ((farnd? exp) (let ((dasp (make-dasp 'dynamic (eval (farnd-pc exp) E A jp)     ⟵ [3]
                                               (eval (farnd-adv exp) E A jp))))
                        (eval (farnd-body exp) E (cons dasp A) jp)))
        ((fun? exp)   (make-closure (fun-params exp) (fun-body exp) E (collect-static A)))     ⟵ [4]
        ((app? exp)   (let* ((cl   (eval (app-fun exp) E A jp))
                             (args (eval-args (app-args exp) E A jp))
                             (njp  (make-jp cl args jp))     ⟵ [5]
                             (env  (extend-env (closure-params cl) args (closure-env cl)))
                             (asps (append (collect-dynamic A) (closure-aspects cl))))     ⟵ [6]
                         (weave-all A njp)     ⟵ [7]
                         (eval (closure-body cl) env asps njp)))     ⟵ [8]
        ...))
(define (collect-static asps)  (collect-if (lambda (a) (eq? (dasp-scope a) 'static)) asps))
(define (collect-dynamic asps) (collect-if (lambda (a) (eq? (dasp-scope a) 'dynamic)) asps))
```

Figure 5. Simplified AspectScheme interpreter.

Reciprocally, the aspect environment in which the body of the function is to be evaluated, A_{app}, is the union of *(a)* the aspects with dynamic scope in the current aspect environment, and *(b)* the aspect environment captured in the closure $\boxed{6}$:

$$A_{app} = \{\langle s, pc, adv\rangle \in A \mid s = \texttt{dynamic}\} \cup closure.A$$

Finally, to implement before advice weaving, the interpreter creates the new join point corresponding to the function application $\boxed{5}$ and triggers weaving before the actual evaluation of the body of the applied function[6]. `weave-all` applies the weaving function to *all* the aspects in the current environment $\boxed{7}$. Therefore, A_{weave}, the set of aspects in the current environment that are woven at the new join point, is simply defined in AspectScheme as $A_{weave} = A$.

Finally, once the advices of the matching aspects have been evaluated, the interpreter evaluates the function body in the corresponding environments, with the new join point as parameter $\boxed{8}$.

To sum up, the essential part of the semantics of statically- and dynamically-scoped aspects in AspectScheme can be described by the structure of the aspect environment A, and the definition of A_{def}, A_{app}, and A_{weave} (Fig. 6).

From functions to objects. These scoping semantics can be formulated in an object-oriented context as follows. With objects, the call stack is modified by the evaluation of a method call (message send). A_{call}, the aspect environment that is used to evaluate the body of a method, is equivalent to A_{app}. Similarly to closures, objects carry along their aspect environment. A_{new}, the aspect environment that is captured within a newly-created object, is equivalent to A_{def}. A_{weave} is unaffected by the paradigm shift. So, the AspectScheme semantics transposed to an object-oriented language are the same as those of Fig. 6, replacing $closure$ by $this$, which refers to the currently-active object. In the following, whenever A_{new} and A_{call} are equivalent to A_{def} and A_{app}, respectively, we do not make the distinction between both.

[6] Here, dynamically-scoped aspects are applied before statically-scoped ones. This arbitrary choice reflects the fact that static aspects are engrained within the function, and can therefore be considered "part of" (or at least "closer to") the original function definition. Also arbitrarily, a deployed aspect is added at the beginning of the environment, giving it default precedence over the previously-deployed ones. We do not discuss aspect composition any further in this paper.

$$\begin{aligned}
A &= \{\langle s, pc, adv\rangle \mid s \in \mathcal{S},\ pc \in \mathcal{PC},\ adv \in \mathcal{ADV}\} \\
A_{def} &= \{\langle s, pc, adv\rangle \in A \mid s = \texttt{static}\} \\
A_{app} &= \{\langle s, pc, adv\rangle \in A \mid s = \texttt{dynamic}\} \cup closure.A \\
A_{weave} &= A
\end{aligned}$$

Figure 6. AspectScheme semantics in a nutshell.

6. Semantics of Deployment Strategies

The unidimensional characterization of scoping provided by AspectScheme does not suffice to express the examples of Sect. 3. The problem is that a dynamically-scoped aspect is *always and only* propagated in A_{app}, and a statically-scoped aspect is *always and only* propagated in A_{def}. The discussed deployment scenarios require much more flexibility.

To address this, we introduce deployment strategies, which support expressive scoping of dynamically-deployed aspects (Sect. 4). A deployment strategy accompanies an aspect instance during its deployment, and describes the scoping semantics associated to its life cycle while deployed. The determining events in the life cycle of a deployed aspect can be seen from Fig. 5:

(i) the aspect is propagated on the call stack $\boxed{6}\boxed{8}$;
(ii) the aspect is propagated in a closure $\boxed{4}$ or object;
(iii) the aspect is woven at the new join point $\boxed{7}$.

Since we aim at allowing any combination of these dimensions, a deployment strategy $\delta\langle c, d, f\rangle$ has three components, each one addressing a particular dimension, and corresponding to the definitions of the three environments A_{def}, A_{app} and A_{weave}. The deployment expression $depl(a, \delta\langle c, d, f\rangle, e)$ deploys the aspect a on the expression e with the semantics specified by the strategy δ.

We therefore extend the representation of an aspect environment, *i.e.* a set of *deployed* aspects, as follows:

$$A = \{\langle a, \delta\langle c, d, f\rangle\rangle \mid a \in \mathcal{ASP},\ c, d, f \in \mathcal{PC}\}$$

where $\mathcal{ASP}$ is the type of aspects (how aspects are precisely defined is not relevant for our purposes).

deployment	scoping semantics
$\delta\langle false, true, \_\rangle$	statically-scoped aspects
$\delta\langle true, false, \_\rangle$	dynamically-scoped aspects
$\delta\langle true, true, \_\rangle$	all join points in dynamic extent as well as in unapplied functions
$\delta\langle false, false, \_\rangle$	only lexically-visible join points that are immediately evaluated

Table 1. Scoping semantics with propagation constants.

Note that call stack propagation is thread-local: the deployed aspect affects only the join points produced by the activity of the thread executing the deployment block. Conversely, delayed evaluation is thread-global: the aspect sees all join points produced by the execution of the procedural value (function or object), no matter which thread is active.

In the following, we describe more precisely the components c and d, used to control aspect propagation, and the component f, used for filtering out join points.

6.1 Controlling Propagation

We first discuss propagation functions. We assume the mapping to object-oriented languages discussed previously; Sect. 6.1.3 discusses the details specific to propagation with objects.

6.1.1 Constant-Valued Propagation Functions

Let us first consider the case where propagation functions are constant-valued. That is, c (for call stack propagation), and d (for delayed evaluation propagation) are either the $true$ or $false$ constant functions. In this model, the aspect environments A_{def} and A_{app} (recall Sect. 5.2) become:

$$A_{def} = \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid d = true\}$$
$$A_{app} = \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid c = true\} \cup closure.A$$

When a function is defined, the closure captures only those aspects whose propagation function d is the $true$ function; when a function is applied, the body is evaluated in an aspect environment comprised of the aspects in the current aspect environment whose propagation function c is $true$, plus the aspects captured in the closure about to be applied.

This simple model makes it possible to express four scoping semantics, two of which correspond to the statically- and dynamically-scoped aspects of AspectScheme (Table 1). The third alternative, $\delta\langle true, true, \_\rangle$, solves Cases 1 and 2 in Sect. 3, in particular the refactoring issue: the deployed aspect affects the evaluation of any function (resp. object) escaping the deployment block, no matter how deep in the call stack it was defined (resp. created). The fourth alternative makes it possible to denote only the lexically-visible and immediately-evaluated join points; this strategy is not possible with the AspectScheme model.

6.1.2 Non-Constant Propagation Functions

With constant-valued propagation functions, if an aspect propagates, it will always do so. Propagation functions in general allow for more flexibility in this regard, by giving the possibility of subjecting propagation to a dynamic condition. On each execution step where the propagation of an aspect is questioned, the propagation function is evaluated, passing it the join point as parameter. That is, c and d are pointcuts, *i.e.* elements of $\mathcal{PC}$.

The actual join point that is passed to the propagation function is the newly-created join point (njp), which corresponds to the expression being interpreted. In this model, the aspect environments A_{def} and A_{app} are defined as follows:

$$A_{def} = \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid d(njp)\}$$
$$A_{app} = \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid c(njp)\} \cup closure.A$$

While most aspect languages that extend object-oriented languages provide object creation join points, AspectScheme does *not* create join points for function definition expressions. Therefore, there is no newly-created join point when evaluating d: the join point passed to the propagation function is the current join point.

This model permits one to express advanced strategies by characterizing the join points upon which an aspect should stop its propagation. This was illustrated in Case 3 (Sect. 4), where call stack propagation is stopped when reaching a facade object, and delayed evaluation propagation is limited to objects of a certain type.

6.1.3 Propagation with Objects

The definition of A_{new} and A_{call} are a direct transposition of A_{def} and A_{app} introduced above:

$$A_{new} = \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid d(njp)\}$$
$$A_{call} = \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid c(njp)\} \cup this.A$$

We now discuss peculiarities of object-oriented languages that have an impact on the aspect propagation model.

Object creation. For a deployed aspect to be captured within the aspect environment of an object A_{new}, it needs to be in the current aspect environment A at the time the object is *effectively created*. In a classless language with ex-nihilo creation of objects like Self [35], as well as in a language with first-order classes, like Java, object creation is done via a particular expression (`new`): the precise locus of creation is therefore straightforward to localize.

Conversely, in a language with first-class classes, like Smalltalk, there is no dedicated expression: objects are instantiated by standard message sending. The message is processed by the class, and at some point, possibly after processing various messages, the class may invoke a creation primitive, like `basicNew`, resulting in the actual allocation of the new object. Therefore, to be captured in an object, an aspect should be deployed so as to propagate on the stack up to that point, at least for class-level messages (indeed, this distinction can be done by the call stack propagation function).

Self sends and super sends. In object-oriented programming, some message sends are peculiar, because they are sent to the currently-active object: self sends and super sends. Because these are method calls affecting the call stack, the call stack propagation function of an aspect is used to determine whether the aspect propagates with the method call or not.

There is a particular case to consider: if an aspect is engrained within the currently-active object, it does see all join points occurring within methods executing on that object, even if it does not propagate on the stack. This can be seen from the definition of A_{call} above: if an aspect a has been captured in the aspect environment of the object (*i.e.* it is part of $this.A$), it is included in A_{call}, regardless of the call stack propagation function of its deployment.

6.2 Join Point Filtering

Beyond the components c and d related to the propagation of an aspect, there is a third component to a deployment strategy: a join point filter f. Such a filter is a *deployment-local* refinement of the pointcuts of an aspect. As a result, the aspect sees *less* join points.

Recall from Sect. 5.2 (Fig. 6) that in AspectScheme, as well as in the other deployment mechanisms we are aware of, the set of aspects that is considered for weaving at a new join point, A_{weave}, is the current aspect environment A. The f component of a deployment strategy makes it possible to introduce deployment control

$$
\begin{aligned}
A &= \{\langle a, \delta\langle c, d, f\rangle\rangle \mid a \in \mathcal{ASP},\ c, d, f \in \mathcal{PC}\} \\
A_{def} &= \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid d(njp)\} \\
A_{app} &= \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid c(njp)\} \cup closure.A \\
A_{weave} &= \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid f(njp)\}
\end{aligned}
$$

Figure 7. Deployment strategies semantics in a nutshell.

over A_{weave}. Like propagation functions, filters are boolean predicates parameterized by the new join point. That is, $f \in \mathcal{PC}$. In this model, A_{weave} is defined as:

$$A_{weave} = \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid f(njp)\}$$

The introduction of join point filters makes it possible to solve Case 4 as shown in Sect. 4, where a filter is used to limit the activity of an aspect to those join points where dynamically-determined objects (cars with the security pack option) are involved.

Join point filters and propagation functions serve really orthogonal objectives. While propagation functions delimit the scope boundaries of an aspect, join point filters act as a means to hide certain join points occurring within those boundaries. When a propagation function returns false, the aspect is never propagated along that dimension anymore. On the contrary, if a join point filter rules out a join point, the aspect is still active, and can continue matching join points and propagating as specified by the other components of the deployment strategy.

To sum up, the semantics of deployment strategies can be described by the new structure of an aspect environment A, and the definitions of the environments at determining points in the interpretation process. This is shown on Fig. 7, and can be contrasted with the summary of AspectScheme semantics, Fig. 6.

7. Interpretation of Deployment Strategies

We now describe the semantics of our model using an interpreter-based operational description. First, we revisit the AspectScheme interpreter of Fig. 5, thereby clarifying the semantics of deployment strategies in a higher-order functional language. Then, we expose a variant of the Aspect SandBox interpreter of Masuhara *et al.* [25], to show the semantics of our model in an object-oriented language.

7.1 Functional Base Language

We now provide the operational semantics of deployment strategies in a functional language, via the Scheme interpreter of Fig. 8. Compared to the AspectScheme interpreter (Fig. 5), `around` and `fluid-around` expressions have been replaced by `depl`. When interpreting a deployment expression, the interpreter creates a deployed aspect `dasp`, which aggregates the deployment strategy $\delta\langle c, d, f\rangle$ and the aspect itself $\boxed{9}$. It then evaluates the body of the deployment expression within the extended aspect environment $\boxed{10}$.

When a function definition is evaluated, only the aspects in the current environment that should be propagated in the dimension of delayed evaluation are captured in the closure $\boxed{11}$. `collect-match-d` returns the list of all given aspects for which the d propagation function applied to the current join point yields true $\boxed{15}$. This defines A_{def}.

When evaluating a function application, the interpreter appends the set of aspects that should be propagated in the call stack (that is, whose c propagation function matches the newly-created join point `njp`), to the aspects previously captured in the closure $\boxed{12}$. This defines A_{app}.

When a new join point is created, instead of `weave-all`, the interpreter calls `weave-some` $\boxed{13}$: only the aspects whose join point filter matches the new join point are considered for weaving $\boxed{14}$. This defines A_{weave}.

```
(define (eval exp E A jp)
 (cond
  ((const? exp) (const-value exp))
  ((var? exp)   (lookup (var-name exp) E))
  ((depl? exp)                                           16
   (let ((dasp (make-dasp exp E A jp)))
     (eval (depl-body exp) E (cons dasp A) jp)))
  ((new? exp)
   (let* ((class (lookup-cls (new-class exp)))
          (args  (eval-args (new-args exp) E A jp)))
     (new-obj class args A jp)))
  ((call? exp)
   (let* ((sig  (call-signature exp))
          (obj  (eval (call-target exp) E A jp))
          (args (eval-args (call-args exp) E A jp)))
     (call-method sig obj args A jp)))
  ...))

(define (new-obj class args asps jp)
 (let* ((njp (make-jp 'new #f class args jp)))          17
   (weave-some asps njp)                                 18
   (let* ((vals   ...create vector for fields)
          (nasps  (collect-match-d njp A))               19
          (obj    (make-object class vals nasps)))       20
    (execute (lookup 'init class) obj args asps njp)     21
    obj)))

(define (call-method sig obj args asps jp)
  (let ((njp (make-jp 'call sig obj args jp)))           22
    (weave-some asps njp)                                23
    (execute (lookup-method sig (object-class obj))
             obj args asps njp)))

(define (execute method this args asps jp)
  (let* ((env ...new environment that includes bindings
               ...for this, declaring class, and args)
         (nasps (union (collect-match-c jp asps)         24
                       (object-asps this)))              25
         (njp (make-jp 'exec (method-name method)
                       this args jp)))                   26
    (weave-some nasps njp)                               27
    (eval-body (method-body method) env nasps jp)))      28
```

Figure 9. Interpretation of deployment strategies for an object-oriented language.

7.2 Object-Oriented Base Language

Within the Aspect SandBox (ASB) project, Masuhara, Kiczales and Dutchyn developed a Scheme interpreter to study concise models of AOP, both for theoretical studies and for prototyping alternative AOP semantics and techniques [25]. We now give the operational semantics of our model of expressive aspect scoping in an object-oriented language as a variation of this interpreter.

The interpreter supports a simplified AspectJ-like language, whose object-oriented features are like a simplified Java. In particular, it supports first-order classes. Since the original ASB interpreter was used to model AspectJ-like aspects, there are two notable differences with our interpreter (Fig. 9): *(a)* the expression language includes an expression for dynamic aspect deployment, `depl`, which has the exact same semantics as in the functional case $\boxed{16}$; *(b)* the interpreter passes aspect environments around, in order to properly implement the scoping semantics, whereas the original ASB interpreter uses a global aspect environment. Also, as previously, for clarity we consider only before advices. To be

```
(define (eval exp E A jp)
  (cond ((const? exp) (const-value exp))
        ((var? exp)   (lookup (var-name exp) E))
        ((depl? exp)  (let ((dasp (make-dasp exp E A jp)))    ⟵ 9
                        (eval (depl-body exp) E (cons dasp A) jp)))    ⟵ 10
        ((fun? exp)   (make-closure (fun-params exp) (fun-body exp) E (collect-match-d jp A)))    ⟵ 11
        ((app? exp)   (let* ((cl   (eval (app-fun exp) E A jp))
                             (args (eval-args (app-args exp) E A jp))
                             (njp  (make-jp cl args jp))
                             (env  (extend-env (closure-params cl) args (closure-env cl)))
                             (asps (append (collect-match-c njp A) (closure-aspects cl))))    ⟵ 12
                        (weave-some A njp)    ⟵ 13
                        (eval (closure-body cl) env asps njp)))
        ...))
(define (weave-some A jp) (weave-all (collect-match-f A jp) jp))    ⟵ 14
(define (collect-match-c jp asps) (collect-if (lambda (a) ((dasp-c a) jp)) asps))
(define (collect-match-d jp asps) (collect-if (lambda (a) ((dasp-d a) jp)) asps))    ⟵ 15
(define (collect-match-f jp asps) (collect-if (lambda (a) ((dasp-f a) jp)) asps))
```

Figure 8. Interpretation of deployment strategies for a higher-order functional language.

precise, we now consider three kinds of join points: `new`, `call`, and `exec`, for object creation, method call and execution, respectively.

A `new` expression provokes the creation of a join point $\boxed{17}$, against which aspects in the current environment are woven $\boxed{18}$. An object is a structure that holds an aspect environment A_{new}: this environment is obtained by collecting all aspects in the environment whose delayed evaluation propagation function matches the new join point $\boxed{19}$. Once created $\boxed{20}$, the constructor of the object is called (modeled as an `init` method) $\boxed{21}$.

When a method call is interpreted, the interpreter creates a `call` join point $\boxed{22}$ and weaves any aspect in the current environment that matches the new join point $\boxed{23}$. It then triggers the execution of the method. This means creating the aspect environment that will be used to evaluate the body of the method, A_{call}: the union of the aspects in the current environment whose call stack propagation function matches the current call join point $\boxed{24}$, and the aspects contained in the aspect environment captured by the now currently-executing object $\boxed{25}$. The interpreter then creates the new execution join point $\boxed{26}$ and weaves aspects in the new environment $\boxed{27}$, before evaluating the method body $\boxed{28}$.

As in the functional case, `weave-some` is used for weaving ($\boxed{18}\boxed{23}$): it first evaluates the join point filters of the aspects in the current environment before invoking `weave` for actual weaving.

8. Relation to Existing Scoping Semantics

We now discuss the relation between our scoping model and existing proposals, including how they can be expressed in our model, and vice-versa.

For the sake of clarity of the comparison, we distinguish between being able to express a particular scoping semantics and being only able to emulate it. A model is able to *express* (or *supports*) a particular scoping semantics for the deployment of aspect a iff the mechanism can be obtained without altering the definition of a, and without altering the expression on which a is deployed.

The first condition is to ensure that the advantage of control and reuse of aspect definitions is maintained; the second is to ensure that the deployment is oblivious to the expression on which it occurs. In contrast, most of the semantics described can be *emulated* in AspectJ, by introducing specific state into the aspect and extra dynamic conditions to its pointcuts. It is clear that this sacrifices at least one of the two above conditions.

8.1 Static Deployment

First of all, we briefly discuss how static aspect deployment, as provided by AspectJ [21], can be expressed. A statically-deployed aspect is an aspect that sees all join points occurring during the entire execution of the program. CaesarJ supports static deployment by means of a keyword to declare an aspect as `deployed`.

In our model, supporting this semantics simply means passing an aspect environment that contains all aspects that should be deployed statically, along with the initial program expression. The propagation semantics of a statically-deployed aspect is that it always propagates on the call stack ($\delta\langle true, \_, \_\rangle$): since any expression is executed in the dynamic extent of the initial program expression, the aspects thus deployed see all join points; delayed evaluation propagation is therefore not required in this case.

8.2 Dynamic Deployment

Among approaches for dynamic deployment of aspects, there are various scoping mechanisms proposed: global, per thread, lexical, and per object.

8.2.1 Global scope

Several aspect languages, like CaesarJ, JAsCo [33], as well as AOP frameworks, like AspectS [19], support a means to globally and dynamically deploy or undeploy an aspect. For instance, in CaesarJ, this mechanism is known as local deployment (local to a JVM): an aspect thus deployed is active from the time it is deployed to the time it is explicitly undeployed, and this for all running threads.

Supporting global scope deployment requires a global aspect environment, shared by all threads. The global aspect environment can be the initial environment discussed above, so that all statically-deployed aspects reside in this environment. In addition, the language must provide two expressions for explicit (un)deployment. For instance, the global deployment expression `gdepl` can be interpreted as follows:

```
((gdepl? exp)
  (let* ((dasp (make-dasp exp E A jp))
    (set! *global-asps* (add dasp *global-asps* asps))))
```

By definition, the propagation functions of the aspects in the global aspect environment are ignored, since the aspects residing there have a global scope.

Douence *et al.* [14] propose an operator for sequential composition of aspects, $A_1 - C \rightarrow A_2$, meaning "A_1 until C (events) then A_2". This operator is a flexible means of delimiting the scope of an aspect based on some event occurrences. In particular, one can define the operators "A until C" and "A from C". These deployments have global scope and can therefore be supported in a similar manner. In its general form, $A_1 - C \rightarrow A_2$, the sequencing operator suggests an interesting extension to our scoping model, whereby propagation functions (resp. join point filters) are not simply boolean predicates, but functions that return the aspect that should be propagated (resp. woven), if any. We defer the discussion of this extended model to a later work.

8.2.2 Thread-local scope

We have already related our proposal to aspect deployment on thread-local scope as provided by CaesarJ and AspectScheme. A simple model with propagation constants is enough to express this semantics (Table 1): the deployed aspect always propagates on the call stack, and never propagates with delayed evaluation: $\delta\langle true, false, \_\rangle$. In our model, propagation functions parameterized over join points allow to express more fine-grained stack propagation strategies, by supporting the specification of conditions over the join points at which stack propagation occurs. This makes it possible to stop propagation beyond a certain point.

Among adaptation mechanisms that have a thread-local scope, as far as we know, only ContextL supports local *undeployment* [11]. ContextL is a language for context-oriented programming that allows a program to be defined in mixin layers that can be dynamically combined, activated, and deactivated. Activation and deactivation of layers is thread-local. However, because ContextL does not include a pointcut language for denoting points in a program execution, it is impossible to specify, at layer *deployment time*, a condition upon which the layer must be deactivated. Deactivation must be done explicitly at all required sites. Interestingly, this deactivation can be specified in a dedicated layer.

8.2.3 Lexical scope

The statically-scoped aspects of AspectScheme have semantics $\delta\langle false, true, \_\rangle$. These aspects propagate along with delayed evaluation, however it is not possible to limit this propagation.

Lexical scoping can be expressed in AspectJ using program text-based pointcuts such as `within` and `withincode`. Beyond the fact that lexical deployment cannot be done dynamically, there is an important difference with the semantics *a la* AspectScheme that we have discussed for objects. In AspectJ, an aspect only propagates to an object created with the lexical scope if the type definition of the object is nested within that scope, *i.e.* it is an inner class.

This lexical scoping rule can be expressed in our model as $\delta\langle false, inner, \_\rangle$, where $inner$ only matches the creation of an inner class. Finally, none of the existing models can express no propagation at all along delayed evaluation.

8.2.4 Object-level scope

In this work, we have come to the notion of aspect environments in objects by elaborating on the lexically-scoped aspects of AspectScheme. Interestingly, the idea of binding an aspect to an object has been proposed from a different perspective, where the focus is on explicitly binding an aspect to an already-created object.

Rajan and Sullivan [31] propose per-object aspects in order to support an aspect-oriented implementation of mediators [32], in the language *Eos*. If an aspect is declared as `instancelevel`, then it only sees the join points occurring within objects that have been explicitly registered using its `addObject` method. An aspect can cease to observe join points in an object using `removeObject`. Per this deployment in CaesarJ is similar, except that the specification of per-object deployment is not mixed with the aspect definition: an aspect is deployed on an object using the runtime library method `deployOnObject` on `DeploySupport`. They can be similarly undeployed by calling `undeployFromObject`. An aspect deployed on an object intercepts only the join points in the execution context of that object. These approaches to per-object aspects are in fact reminiscent from well-known mechanisms in per-object metaobject protocols: a metaobject redefines the semantics of the execution points of its referent [23]. Composition filters are another instantiation of this same model [6].

Our semantics supports per-object aspects, but with two notable differences. First, as in AspectScheme, capturing an aspect in the environment occurs automatically as a result of object creation (function definition), and not explicitly after the object is already created. Supporting explicit per-object deployment is direct:

```
((deplobj? exp)
  (let* ((obj  (eval (deplobj-obj exp) A E jp))
         (dasp (make-dasp exp A E jp)))
    (obj-set-asps! (add dasp (obj-asps obj)))))
```

To interpret the `deplobj` expression, the interpreter first obtains both the object and the aspect, and then adds the aspect in the aspect environment of the object.

The second difference is that in all the above proposals (including metaobjects and per-object aspects), an aspect only sees the join points where the executing object is the one on which it has been deployed. This means that, once deployed on the object, the aspect does not propagate on the stack (*e.g.* to affect the dynamic extent of the activity of the object), nor does it propagate with delayed evaluation (*e.g.* to affect the behavior of objects created therefrom). In other words, the aspect is deployed as $\delta\langle false, false, \_\rangle$. Our model permits a per-object aspect deployment that includes the full specification of the scoping semantics, including both propagation functions and join point filters, once engrained in the object.

8.3 Expressing Propagation Functions

Propagation functions can be used to express very straightforwardly all existing scoping mechanisms, and their expressiveness is superior to each individual mechanism. However, we still have to consider if we can use a *combination* of the existing mechanisms, *e.g.* using deployment aspects, to express propagation functions.

Below, we analyze whether $depl(a, \delta\langle c, d, \_\rangle, e)$ can be expressed by a combination of existing proposals, denoted as follows:

- $depl_s(a, e)$: the statically-scoped aspects of [16], with semantics $\delta\langle false, true, \_\rangle$.
- $depl_d(a, e)$: the dynamically-scoped aspects found in *e.g.* AspectScheme and CaesarJ, with semantics $\delta\langle true, false, \_\rangle$.
- $undepl_d(a, e)$: a thread-local undeployment mechanism, like the one found in ContextL [11].
- $depl_o(a, o)$: per-object deployment, that inserts a in the aspect environment of o with semantics $\delta\langle false, false, \_\rangle$.

Controlling stack propagation. $depl_d(a, e)$ does not provide a means to stop the propagation of a along the call stack. The only alternative is to use an undeployment aspect d_a, that upon occurrences of a join point matched by c, undeploys a. This is the "a until c" operator of Douence *et al.* [14], but in a dynamic and thread-local manner. The only thread-local undeployment mechanism we know of is the layer deactivation of ContextL. Assuming the existence of this mechanism $undepl_d(a, e)$ in a language of the pointcut-advice family, one can express $\delta\langle c, \_, \_\rangle$ by deploying at the same time a and an undeployment aspect ud_a:

$$depl_d([a, ud_a], e)$$
$$ud_a : \forall jp, jp.kind = \texttt{call} \wedge c(jp), undepl_d(a, proceed(jp))$$

Whenever a call join point is matched by c, ud_a undeploys a for the rest of the extent of the execution of the matched join point.

Controlling delayed evaluation propagation. $depl_s(a, e)$ implies unbounded propagation of a along the delayed evaluation dimension of e. There is no way to stop propagation according to a propagation criteria d. The alternative consists in using a (non-trivial) combination of $depl_o$ and $depl_d$.

First, we can deploy using $depl_d$ a deployment aspect d_a with the following semantics:

$$d_a : \forall jp, jp.kind = \texttt{new} \wedge d(jp), depl_o(a, obj)$$

Whenever a `new` join point matches the d criteria, d_a deploys a within the newly-created object obj. However, $depl_o$ deploys a in obj with strategy $\delta\langle false, false, \_\rangle$: a will not be deployed within objects created by obj. To solve this, one needs to recursively deploy d_a in obj:

$$d_a : \forall jp, jp.kind = \texttt{new} \wedge d(jp), depl_o([a, d_a], obj)$$

This solution solves the issue of propagating a within objects created by obj, and therefore improves over $depl_s$ by having propagation semantics $\delta\langle false, d, \_\rangle$. Now, if we want to support propagation of a within objects that are not *directly* created by obj but as a result of a call to another object (*e.g.* a factory, or a first-class class), d_a must also be deployed with stack propagation ($depl_d$) for each method called by obj. This can be done by a second deployment aspect d'_{d_a} that has to be deployed with a:

$$d_a : \forall jp, jp.kind = \texttt{new} \wedge d(jp), depl_o([a, d_a, d'_{d_a}], obj)$$
$$d'_{d_a} : \forall jp, jp.kind = \texttt{call}, depl_d(d_a, proceed(jp))$$

This shows that any deployment strategy related to delayed evaluation propagation that can be expressed in our model, can also be expressed using the two deployment mechanisms $depl_o$ and $depl_d$. However, as shown, this requires non-trivial gymnastics with deployment aspects.

Evaluation. The conclusion of the analysis above is that: *(a)* call stack propagation semantics can be expressed by a combination of $depl_d$ and $undepl_d$, and *(b)* delayed evaluation propagation semantics can be expressed by a combination of $depl_d$ and $depl_o$.

The reader can however imagine how to combine the approach to simulate delayed evaluation propagation with the one presented for stack propagation, in order to express any deployment strategy $\delta\langle c, d, \_\rangle$. Besides the fact that we are not aware of any aspect language supporting the three mechanisms $depl_d$, $undepl_d$ and $depl_o$, the intertwined use of deployment and undeployment aspects is, least to say, not convenient.

8.4 Join Point Filtering

The idea of allowing external extensions to all the pointcuts of certain aspects has first been proposed in Extended AspectJ (EAJ) as proposed by the `abc` team [4], using `global` pointcuts. A global pointcut consists of a type pattern for denoting the aspects to which it applies, and a pointcut expression that is a common conjunct added to the pointcut expressions of all advices of the denoted aspects. Global pointcuts can be used to tailor the scope of existing aspects, *e.g.* by excluding certain classes. The join point filtering mechanism of our model can be seen as a refinement of global pointcuts in the context of dynamic deployment of aspects: the conjunct is applied *only* to the deployed aspect instance, *only* for the extent specified by the other components of the deployment strategy. In contrast, global pointcuts are a static facility, with global scope.

Extending a pointcut at deployment time is trivial in AspectScheme, because a pointcut is a first-class function that can be wrapped as needed. Externally extending an aspect definition can also be done with composition filters [6], however there are no specific provisions to make the extension of a filter limited to a particular scope, either with respect to the call stack or to control its propagation amongst newly-created objects.

9. Conclusion

We propose an expressive scoping model for dynamically-deployed aspects: *deployment strategies*. Deployment strategies provide explicit control over the propagation properties of a deployed aspect, both along call stack and delayed evaluation dimensions, as well as deployment-specific join point filters. We have formulated deployment strategies for both functional and object-oriented based aspect languages.

This work provides a common operational framework for understanding the relation between various scoping semantics. In particular, we have clarified the relation between statically-, dynamically-scoped and per-object aspects as provided by existing systems. But deployment strategies allow the exploration of a much wider design space for scoping, as has been illustrated.

Expressive scoping of dynamically-deployed aspects fosters more reusable aspect definitions, while providing very fine-grained control over scoping for specific deployment scenarios. Such a control is crucial to avoid unexpected aspect effects, in particular when using independently-developed aspects [26]. Still, it is clear that scoping alone does not suffice to address the inherent complexity of aspect-oriented software development, and that development methodologies, design patterns, dedicated syntactic constructs and appropriate debugging tools remain crucial.

There are various possible extensions to the model we presented, by considering language features that can introduce extra dimensions for scoping: exceptions, virtual classes, as well as concurrent and distributed AOP. Also, our treatment of deployment strategies focus on behavioral adaptation with the pointcut-advice mechanism. Many interesting aspects actually rely on both behavioral and structural adaptation [1], *e.g.* using intertype declarations. There exist several proposals for scoping structural adaptations, using either static scoping [5, 37] or dynamic scoping [11]. Similarly to this work, we are interested in studying a common operational framework for exploring the scoping design space of structural adaptation.

Acknowledgments. For very valuable input on this work and its presentation, we thank Pascal Costanza, Chris Dutchyn, Johan Fabry, Kris Gybels, Shriram Krishnamurthi, Olivier Motelet, Jacques Noyé, and Guillaume Pothier, as well as the anonymous reviewers.

References

[1] Sven Apel, Christian Kstner, and Salvador Trujillo. On the necessity of empirical studies in the assessment of modularization mechanisms for crosscutting concerns. In *1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM.07)*, Minneapolis, Minnesota, USA, May 2007. IEEE Computer Society Press. In conjunction with ICSE 2007.

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, February 2006.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 117–128, Chicago, USA, June 2005.

[4] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible

AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag, 2006.

[5] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2005)*, pages 177–189, San Diego, California, USA, October 2005. ACM Press. ACM SIGPLAN Notices, 40(11).

[6] Lodewijk Bergmans and Mehmet Akşit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[7] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting virtual machine techniques for seamless aspect support. In OOPSLA 2006 [30], pages 109–124.

[8] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In OOPSLA 2006 [30], pages 125–138.

[9] Christoph Bockish, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In Lieberherr [22], pages 83–92.

[10] Eric Bodden, Laurie Hendren, and Ondrej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In Erik Ernst, editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in Lecture Notes in Computer Science, pages 525–549, Berlin, Germany, july/august 2007. Springer-Verlag.

[11] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming – an overview of ContextL. In *ACM Dynamic Language Symposium (DLS 2005)*, San Diego, CA, USA, October 2005.

[12] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient layer activation for switching context-dependent behavior. In *Proceedings of the Joint Modular Languages Conference (JMLC 2006)*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103, Oxford, England, September 2006. Springer-Verlag.

[13] Bruno De Fraine and Mathieu Braem. Requirements for reusable aspect deployment. In Markus Lumpe and Wim Vanderperren, editors, *Proceedings of the 6th International Symposium on Software Composition (SC 2007)*, number 4829 in Lecture Notes in Computer Science, Braga, Portugal, March 2007. Springer-Verlag.

[14] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [22], pages 141–150.

[15] Christopher Dutchyn. *Dynamic Join Points: Model and Interactions*. PhD thesis, University of British Columbia, Canada, November 2006.

[16] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, December 2006.

[17] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[18] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr [22], pages 26–35.

[19] Robert Hirschfeld. AspectS – aspect-oriented programming with Squeak. In Mehmet Akşit, Mira Mezini, and R. Unland, editors, *International Conference NetObjectDays on Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, 2002.

[20] Robert Hirschfeld and Pascal Costanza. Extending advice activation in AspectS. In *2nd European Interactive Workshop on Aspects in Software (EIWAS 2005)*, Brussels, Belgium, September 2005.

[21] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[22] Karl Lieberherr, editor. *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 2004. ACM Press.

[23] Pattie Maes. Concepts and experiments in computational reflection. In Meyrowitz [27], pages 147–155. ACM SIGPLAN Notices, 22(12).

[24] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In Luca Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 2–28, Darmstadt, Germany, July 2003. Springer-Verlag.

[25] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[26] Nathan McEachen and Roger T. Alexander. Distributing classes with woven concerns – an exploration of potential fault scenarios. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 192–200, Chicago, Illinois, USA, March 2005. ACM Press.

[27] Norman Meyrowitz, editor. *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87)*, Orlando, Florida, USA, October 1987. ACM Press. ACM SIGPLAN Notices, 22(12).

[28] Mira Mezini and Klaus Ostermann. Object creation aspects with flexible aspect deployment. Technical report, Technische Universität Darmstadt, 2003.

[29] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th Symposium on Foundations of Software Engineering (FSE-12)*, pages 127–136, Newport Beach, CA, USA, November 2004.

[30] *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, Portland, Oregon, USA, October 2006. ACM Press.

[31] Hridesh Rajan and Kevin Sullivan. Eos: Instance-level aspects for integrated system design. In *Proceedings of ESEC/FSE 2003*, pages 297–306, Helsinki, Finland, September 2003.

[32] Kevin Sullivan and D. Notking. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[33] Davy Suvee, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In Mehmet Akşit, editor, *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 21–29, Boston, MA, USA, March 2003. ACM Press.

[34] Naoyasu Ubayashi, Genki Moriyama, Hidehiko Masuhara, and Tetsuo Tamai. A parameterized interpreter for modeling different AOP mechanisms. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 194–203, Long Beach, CA, USA, 2005. ACM Press.

[35] David Ungar and Randall B. Smith. Self: The power of simplicity. In Meyrowitz [27], pages 227–241. ACM SIGPLAN Notices, 22(12).

[36] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

[37] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In OOPSLA 2006 [30], pages 37–55.