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].

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.

---

* Partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and FONDECYT project 11060493.
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 \( \mathcal{PC} = \text{JoinPoint} \rightarrow \text{Bool} \). A call pointcut designator (i.e. a function that returns a pointcut) can be defined as follows:

\[
\begin{align*}
\text{(define call} & \quad (\lambda (f) (\lambda (jp) (eq? f (car jp))))))\\
\text{If open-file is a function in scope, (call open-file) returns a pointcut that matches application of that function. An advice is a join point transformer, that takes a join point as parameter and returns a new function to use instead. It is a function of type} & \quad \mathcal{ADV} = \text{JoinPoint} \rightarrow \text{Val} \rightarrow \text{Val} \text{, e.g.}\\
\text{(define trace} & \quad (\lambda (jp) (\lambda (arg) (printf \text{ "calling"})))\\
\text{The trace advice, for a join point jp, returns a function that when applied to the argument, emits a trace and performs the original function application. app/prim is similar to proceed in AspectJ, and ensures that applying jp will not invoke any further aspects.}
\end{align*}
\]

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 \( \mathcal{PC} \rightarrow \mathcal{ADV} \rightarrow \text{Exp} \rightarrow \text{Val} \)):

- 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.

\[\text{(let ((app/prim (lambda (f) (f "brussels")))}}\]

\[\text{(fluid-around (call open-file) trace (apply-to-brussels open-file))}}\]

Above, the \text{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 \text{apply-to-brussels to open-file}. When \text{open-file} is finally applied (the framebox above), the trace aspect \text{does see} the join point, because the application occurs in the dynamic extent of the fluid-around body expression. Conversely, in:

\[\text{(let ((traced-open (fluid-around (call open-file) trace (lambda (f) (open-file f))))))}}\]

\[\text{(traced-open "brussels")}}\]

The aspect does not apply, because the dynamic extent of the aspect body only consists of a function definition. The (framed) application of \text{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:

\[\text{(let ((traced-open (around (call open-file) trace (lambda (f) (open-file f))))))}}\]

\[\text{(traced-open "brussels")}}\]

The aspect applies, because the application of \text{open-file} occurs \text{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 \text{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-advise 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()
        QAReport qa = (...)? new QALevel1() : new QALevel0();
        deploy-2(qa){
            ProdLine.createAndDispatch(...);
        }
}

class CarFactory {
    Bill order(...) {
        QAReport qa = (...)? new QALevel1() : new QALevel0();
        deploy-2(qa){
            ProdLine.createAndDispatch(...);
        }
    } }

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 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

170
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
of the relevant production steps. This is exactly like
the deployment strategy
depl

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 as-
   pect be captured in created procedural values (functions, ob-
   jects) 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 propa-
   gation properties? or should all the pointcuts of the deployed
   aspect possibly be refined in a deployment-local manner, leav-
   ing the aspect definition unchanged?

The problem of existing scoping models for dynamic aspect de-
ployment is that (a) they lack fine-grained control over dimensions
1 and 2, (b) they do not make it possible to express custom combi-
nations 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 deploy-
ment strategy \( \delta(c, d, f) \) specifies the scoping semantics of a
dynamically-deployed aspect \( a \) with three components, corre-
ponding 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 propa-
  gation 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 \( P(C) \).

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

\[
deploy(a, \delta(c, d, f), e)
\]

\( deploy \) deploys aspect \( a \) (or a list of aspects) on expression \( e \)
using the deployment strategy \( \delta \). When it comes to illustrating our model
with a Java-like language, we use the following syntax, an exten-
sion 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
\( _\delta \equiv _\) 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, 1(qa)]
\[
\text{ProdLine.createAndDispatch(...)};
\]

With this deployment strategy \( \delta(true, true, _) \), 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 QA Site.

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), 1(qa)]
\[
\text{ProdLine.createAndDispatch(...)};
\]

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, true, \{\text{c}(\text{args}(i))\}](\text{sp})
\[
\text{next.process(batch)};
\]

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 \( \text{jp.args}(1) \)) is a car for which the security pack applies (that is, a car contained in the \text{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 \textit{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 of 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, advice 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\(^5\) 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 \( \text{const?} \), and accessed with accessors such as \( \text{const-value} \).

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 \textit{aspect environment} that is passed around between evaluation steps\(^6\), 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:

\[
A = \{ (s, pc, adv) \mid s \in S, pc \in PC, adv \in ADV \}
\]

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

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

When evaluating an aspect deployment expression, the interpreter extends the current aspect environment with the deployed aspect \( \text{aspect} \), and evaluates the body (eq. \( 4 \)). In the case of an \textit{around} expression, corresponding to the deployment of a statically-scoped aspect, the aspect is marked with the \textit{static} attribute\(^7\) if the expression is a \textit{fluid-around}, the aspect scope is set to \textit{dynamic}\(^8\).

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 \( A \).

\[
A_{def} = \{ (s, pc, adv) \in A \mid s = \text{static} \}
\]

\(^5\)The executable Scheme interpreters, along with examples, are available at: http://pleiad.dcc.uchile.cl/research/scope

\(^6\)The ASB interpreter of [25] and the formal model of [36] use a global aspect environment, which is insufficient for modeling \textit{a la} AspectScheme [16] (and by extension, deployment strategies as well).
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 $\circ_A$:

$$A_{app} = \{ (s, pc, adv) \in A \mid s = \text{dynamic} \} \cup \text{closure}.A$$

Finally, to implement before advice weaving, the interpreter creates the new join point corresponding to the function application $\circ_A$ and triggers weaving before the actual evaluation of the body of the applied function $\circ_A$. $\text{weave-all}$ applies the weaving function to all the aspects in the current environment $\circ_A$. 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 matches of the matching aspects have been evaluated, the interpreter evaluates the function body in the corresponding environments, with the new join point as parameter $\circ_A$.

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 $\text{closure}$ 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.

\footnote{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.}

\[ A = \{ (s, pc, adv) \mid s \in S, pc \in PC, adv \in ADV \} \]
\[ A_{def} = \{ (s, pc, adv) \in A \mid s = \text{static} \} \]
\[ A_{app} = \{ (s, pc, adv) \in A \mid s = \text{dynamic} \} \cup \text{closure}.A \]
\[ A_{weave} = A \]

**Figure 5.** Simplified AspectScheme interpreter.

**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 $\circ_A$;
(ii) the aspect is propagated in a closure $\circ_A$ or object;
(iii) the aspect is woven at the new join point $\circ_A$.

Since we aim at allowing any combination of these dimensions, a deployment strategy $\delta(c, d, f)$ 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 $\text{depl}(a, \delta(c, d, f), c)$ deploys the aspect $a$ on the expression $c$ with the semantics specified by the strategy $\delta$.

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

$$A = \{ (a, \delta(c, d, f)) \mid a \in ASP, c, d, f \in PC \}$$

where $ASP$ is the type of aspects (how aspects are precisely defined is not relevant for our purposes).
Let us first consider the case where propagation functions are constant-valued. In this model, the aspect environments \( A_{\text{def}} \) and \( A_{\text{app}} \) (recall Sect. 5.2) become:

\[
A_{\text{def}} = \{(a, \delta(c, d, f)) \in A \mid d = \text{true}\}
\]

\[
A_{\text{app}} = \{(a, \delta(c, d, f)) \in A \mid c = \text{true}\} \cup \text{closure}.A
\]

When a function is defined, the closure captures only those aspects whose propagation function \( d \) is the \text{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 \text{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, \( \delta(\text{true, true, ...}) \), solves Cases 1 and 2 in Sect. 3, in particular the refactoring issue: the deployed aspect affects the join points produced by the activity 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 now discuss peculiarities of object-oriented languages that have an impact on the aspect propagation model.

#### 6.1.1 Constant-Valued Propagation Functions
Let us 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 \text{true} or \text{false} constant functions. In this model, the aspect environments \( A_{\text{def}} \) and \( A_{\text{app}} \) introduced above:

\[
A_{\text{def}} = \{(a, \delta(c, d, f)) \in A \mid d(njp)\}
\]

\[
A_{\text{app}} = \{(a, \delta(c, d, f)) \in A \mid c(njp)\} \cup \text{closure}.A
\]

### 6.1.3 Propagation with Objects
The definition of \( A_{\text{new}} \) and \( A_{\text{call}} \) are a direct transposition of \( A_{\text{def}} \) and \( A_{\text{app}} \) introduced above:

\[
A_{\text{new}} = \{(a, \delta(c, d, f)) \in A \mid d(njp)\}
\]

\[
A_{\text{call}} = \{(a, \delta(c, d, f)) \in A \mid c(njp)\} \cup \text{this}.A
\]

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 \( A_{\text{new}} \), it needs to be in the current aspect environment \( A \) at the time the object is \text{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 \text{basically new}, 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 engrafted 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 \( A_{\text{call}} \) above: if an aspect \( a \) has been captured in the aspect environment of the object (i.e. \( a \) is part of \( \text{this}.A \)), it is included in \( A_{\text{call}} \), 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, \( A_{\text{scope}} \), is the current aspect environment \( A \). The \( f \) component of a deployment strategy makes it possible to introduce deployment control.
over $A_{\text{weave}}$. Like propagation functions, filters are boolean predicators parameterized by the new join point. That is, $f \in \mathcal{PC}$. In this model, $A_{\text{weave}}$ is defined as:

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

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 $\text{depl}$. When interpreting a deployment expression, the interpreter creates a deployed aspect $\text{dasp}$, which aggregates the deployment strategy $\delta(c, d, f)$ and the aspect itself. It then evaluates the body of the deployment expression in the extended aspect environment.

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. The $\text{collect-match-d}$ returns the list of all given aspects for which the $d$ propagation function applied to the current join point yields true. This defines $A_{\text{depl}}$.

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. This defines $A_{\text{app}}$.

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

#### 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: (a) the expression language includes an expression for dynamic aspect deployment, $\text{depl}$, which has the exact same semantics as in the functional case; (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
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 \( \text{new} \) against which aspects in the current environment are woven. An object is a structure that holds an aspect environment \( A_{\text{new}} \): this environment is obtained by collecting all aspects in the environment whose delayed evaluation propagation function matches the new join point \( \text{new} \). Once created, the constructor of the object is called (modeled as an init method) \( \text{init} \).

When a method call is interpreted, the interpreter creates a call join point \( \text{call} \) and weaves any aspect in the current environment that matches the new join point \( \text{call} \). 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, \( A_{\text{call}} \): the union of the aspects in the current environment whose call stack propagation function matches the current call join point \( \text{call} \) and the aspects contained in the aspect environment captured by the now currently-executing object \( \text{call} \). The interpreter then creates the new execution join point \( \text{exec} \) and weaves aspects in the new environment before evaluating the method body \( \text{exec} \).

As in the functional case, weave-some is used for weaving (18-23): 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 \( \delta(\text{true, } \ldots) \); 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 \( \text{gdeploy} \) can be interpreted as follows:

\[
\begin{align*}
\text{gdeploy} \text{ exp} & \to \text{let* } ((\text{dasp} \text{ make-dasp exp E A jp})) \\
& \quad \text{set! global-asps* (add dasp gglobal-asps* asps))}
\end{align*}
\]

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, \( A_1 \rightarrow C \rightarrow A_2 \), 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: \( \delta(\text{true}, \text{false}, \_\)\). 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 \( \delta(\text{false}, \text{true}, \_\)\). 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 \text{within} and \text{withincode}. Beyond the fact that lexical deployment cannot be done dynamically, there is an important difference with the semantics \( \text{a} \) of 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, \text{i.e.} it is an inner class.

This lexical scoping rule can be expressed in our model as \( \delta(\text{false}, \text{inner}, \_\)\), where \text{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 \text{instancelevel}, then it only sees the join points occurring within objects that have been explicitly registered using its \text{addObject} method. An aspect can cease to observe join points in an object using \text{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 \text{deployOnObject} on \text{DeploySupport}. They can be similarly un-deployed by calling \text{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 metaclass protocols: a metaclass 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:

\[
\begin{align*}
\text{(\text{deployobj} e \_\_\_\_\_\_)} &= \text{(let* ((obj (eval (deployobj-obj e) A E jp))} \\
& \text{ (dasp (make-dasp e A E jp)))} \\
& \text{(obj-set-asaps! (add dasp (obj-asaps obj))))}
\end{align*}
\]

To interpret the \text{deployobj} 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 (\text{e.g.} to affect the dynamic extent of the activity of the object), nor does it propagate with delayed evaluation (\text{e.g.} to affect the behavior of objects created therefrom). In other words, the aspect is deployed as \( \delta(\text{false}, \text{false}, \_\)\). 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 engraved 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, \text{e.g.} using deployment aspects, to express propagation functions.

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

- \text{depl}(a, e): the statically-scoped aspects of [16], with semantics \( \delta(\text{false}, \text{true}, \_\)\).
- \text{depl}(a, e, c): the dynamically-scoped aspects found in \text{e.g.} AspectScheme and CaesarJ, with semantics \( \delta(\text{true}, \text{false}, \_\)\).
- \text{undep}(a, e, c): a thread-local undeployment mechanism, like the one found in ContextL [11].
- \text{depl}(a, o): per-object deployment, that inserts a in the aspect environment of o with semantics \( \delta(\text{false}, \text{false}, \_\)\).

Controlling stack propagation. \text{depl}(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 \text{ud}, that upon occurrences of a join point matched by \text{c}, undeploys a. This is the “a until c” operator of Douence \text{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 \text{undep}(a, e) in a language of the pointcut-advise family, one can express \( \delta(c, \_\_\_\_\_) \) by deploying at the same time \text{a} and an undeployment aspect \text{ud}: 

\[
\text{depl}(a, ud, c) = \text{call \_\_\_\_\_\_} \land \text{c(jp), undep}(a, \text{proceed(jp)})
\]

Whenever a \text{call} join point is matched by \text{c}, \text{ud} undeploys \text{a} for the rest of the extent of the execution of the matched join point.
Controlling delayed evaluation propagation. $\text{depl}_d(a, e)$ 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 $\text{depl}_d$ and $\text{undepl}_d$.

First, we can deploy using $\text{depl}_d$ a deployment aspect $d_a$ with the following semantics:

$$d_a : \forall jp. \text{jp.kind} = \text{new} \land d(jp), \text{depl}_d(a, obj)$$

Whenever a new join point matches the $d$ criteria, $d_a$ deploys $a$ within the newly-created object $obj$. However, $\text{depl}_d$ deploys $a$ in $obj$ with strategy $\delta(\text{false}, \text{false}, \_)$: $a$ will not be deployed within objects created by $obj$. To solve this, one needs to recursively deploy $d'_a$ in $obj$:

$$d_a : \forall jp. \text{jp.kind} = \text{new} \land d(jp), \text{depl}_d([a, d_a], obj)$$

This solution solves the issue of propagating $a$ within objects created by $obj$, and therefore improves over $\text{depl}_d$ by having propagation semantics $\delta(\text{false}, d_\_)_$. 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 ($\text{depl}_d$) for each method called by $obj$. This can be done by a second deployment aspect $d'_a$, that has to be deployed with $a$:

$$d_a : \forall jp. \text{jp.kind} = \text{new} \land d(jp), \text{depl}_d([a, d_a], obj)$$

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 $\text{depl}_d$ and $\text{undepl}_d$. 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 $\text{depl}_d$ and $\text{undepl}_d$, and (b) delayed evaluation propagation semantics can be expressed by a combination of $\text{depl}_d$ and $\text{depl}_d$.

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 $\delta(\_\_, d_\_\_, \_\_)_$. Besides the fact that we are not aware of any aspect language supporting the three mechanisms $\text{depl}_d$, $\text{undepl}_d$, and $\text{depl}_d$, the intertwined use of deployment and undepl煜on 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 AspectJ, 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 Dutech, Johan Fabry, Kris Dybels, Shriram Krishnamurthi, Olivier Meote, Jacques Noyé, and Guillaume Pothier, as well as the anonymous reviewers.

References


[4] Pavel Augustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotáč, Ondřej Lhotáč, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Ribble. abc: an extensible...


