Expressive Scoping of Distributed Aspects*

Éric Tanter Johan Fabry
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
http://pleiad.dcc.uchile.cl

Rémi Douence Jacques Noyé
Mario Südholt
Département Informatique
École des Mines de Nantes, France
http://www.emn.fr/x-info/ascola/

ABSTRACT

Dynamic deployment of aspects brings greater flexibility and reuse potential, but requires proper means for scoping aspects. Scoping issues are particularly crucial in a distributed context: adequate treatment of distributed scoping is necessary to enable the propagation of aspect instances across host boundaries and to avoid inconsistencies due to unintentional spreading of data and computations in a distributed system.

We motivate the need for expressive scoping of dynamically-deployed distributed aspects by an analysis of the deficiencies of current approaches for distributed aspects. Extending recent work on deployment strategies for non-distributed aspects, we then introduce a set of high-level strategies for specifying locality of aspect propagation and activation, and illustrate the corresponding gain in expressiveness. We present the operational semantics of our proposal using Scheme interpreters, first introducing a model of distributed aspects that covers the range of current proposals, and then extending it with dynamic aspect deployment. This work shows that, given some extensions to their original execution model, deployment strategies are directly applicable to the expressive scoping of distributed aspects.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

General Terms

Languages, Design

*This work is partially funded by the INRIA/CONICYT project CORDIAL, and the FONDECYT projects 11060493 (É. Tanter) and 1090083 (J. Fabry & É. Tanter).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA.
Copyright 2009 ACM 978-1-60558-442-3/09/03 ...$5.00.

Keywords

Aspect-oriented programming, distribution, dynamic deployment, scope, Scheme, operational semantics.

1. INTRODUCTION

In the pointcut-advice model of aspect-oriented programming [11, 20], as embodied in *e.g.* AspectJ [8], crosscutting behavior is defined by means of pointcuts and advices. A pointcut is a predicate that matches program execution points, called 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. The *scope* of an aspect is the set of join points the aspect *sees*, *i.e.* against which its pointcuts are matched.

A major challenge in aspect language design is to cleanly and concisely express where and when aspects should apply. If aspects potentially see any join point, expressive pointcut languages are the only way to restrict the scope of aspects. However, as repeatedly recognized [1, 5, 12, 16], this can lead to complex pointcut definitions and sacrifices the reuse potential of aspects.

In a distributed system, a new dimension for scoping appears: dealing with the different execution hosts. Distributed AOP can be achieved by combining a normal aspect language like AspectJ with a form of remote procedure call, but this has severe limitations. To tackle these limitations, several distributed aspect languages and frameworks have been proposed [3, 13, 18]. In these proposals, however, aspect deployment is still done statically; leaving the burden of proper scoping to cumbersome pointcut definitions. In addition, if deployment is not done properly, incompatibility errors or unexpected (non-)application of aspects can occur. These issues, along with related work, are further discussed in Section 2.

Expressive scoping of dynamically-deployed aspects, as partially supported by several languages like CaesarJ [1] and AspectScheme [6], is therefore required for distributed aspect-oriented programming. Recently, Tanter has introduced the deployment strategy model [16], which supersedes other proposals by giving programmers very fine-grained control over the scope of an aspect. However, this model does not take distribution into account. This paper explores the issue of expressive scoping of distributed aspects, allowing advanced distributed scoping strategies to be conveniently expressed. Examples are aspects that propagate only on certain hosts, or that "follow" particular objects as they are sent over the network. We further illustrate such scenarios in Section 3.

To address the issue of expressive scoping of distributed aspects, we introduce a set of high-level strategies that specify locality of aspect propagation and activation, complementing the existing proposal of deployment strategies. Section 4 gives an informal presentation of our proposal and shows how distributed deployment strategies concisely express the scenarios considered in Section 3.

Section 5 presents a small operational model of distributed aspects based on a progressively-extended Scheme interpreter. Section 6 dives into deployment strategies in a distributed setting, expounding the semantics of our proposal. Section 7 concludes.

2. THE CASE FOR DYNAMIC DEPLOYMENT OF DISTRIBUTED ASPECTS

The distribution and scoping features of existing models used for aspect-oriented programming of distributed applications can roughly be classified into three categories:

i) No explicit mechanisms for distribution are provided but a local aspect model is used to manipulate an underlying distributed infrastructure. Examples of this category comprise AspectJ [8], applied to RMI-based applications, as well as JBoss AOP and Spring AOP, applied to Enterprise JavaBeans applications. Aspects essentially have global scope in these models, a notable exception being aspects instantiated only on creation of base entities, such as AspectJ's feature for per-target or per-cflow instantiation.

ii) The localization of join points can be explicitly referred to in order to make aspects, in particular pointcut matching, distribution-aware. This is the main characteristics of the models of remote pointcuts [13], Aspects with Explicit Distribution (AWED) [3, 2], DyMAC [9], and ReflexD [18]. Note that, while these models are mostly static, some of them contain dynamic mechanisms (*e.g.* AWED allows host groups to be modified dynamically). In addition to the scoping features of the previous category, these models also partially include explicitly-defined distributed scopes (*e.g.*, deployment on groups of hosts in AWED and ReflexD).

iii) Mechanisms that allow aspects to be deployed on entities of the base application such that the scope of aspects is implicitly limited to occurrences of distributed join points that are generated by the execution of those entities. For example, CaesarJ [1], allows aspects to be deployed dynamically such that they are only triggered within the (distributed) control-flow of certain method calls.

2.1 Issues with Static Deployment

In most aspect languages, aspects are deployed statically, *i.e.* before execution time, and have global scope. Restricting the scope of an aspect can only be done by introducing extra conditions in the pointcut definitions. This however renders pointcut definitions unnecessarily complex and sacrifices the reuse potential of aspects [1, 5, 12, 16]. Moreover, the exact dynamic patterns under which an aspect should be effective may be very hard or impossible to foresee and express statically in the aspect definition.

In a distributed setting, static deployment of aspects with global scope is even more problematic, because the mere notion of "global" is not necessarily straightforward to define. Consider a case where AspectJ is used in conjunction with RMI. Static weaving creates new versions of the potentially impacted classes, which then have to be loaded on all the hosts in a consistent way. If not, a problem occurs when a host has loaded the aspect-free version of a class and receives an instance of the woven version of the same class. Two equally unsatisfactory scenarios are possible: either a class version exception is thrown, or the aspect does not apply[1].

The above issue is not the sole issue: such approaches cannot directly express relationships between different distributed entities [13]. As a consequence, pointcuts that relate join points, like `cflow`, will not work in a distributed setting. To address this, several aspect languages with explicit distribution features have been proposed. These languages are more robust and expressive than a simple combination of an aspect language and a middleware for distribution. DJcutter [13], for instance, introduced the idea of discriminating join points based on their host of occurrence and thus solves the distributed control flow problem; some, like ReflexD [18], give flexible control over the placement of advice instances in the system or, like AWED [2], make it possible to exploit causal orderings between join points on different hosts.

These distributed aspect systems typically offer programmatic means to specify aspect deployment, more convenient than ad-hoc startup-time code. But —apart from CaesarJ, discussed below— deployment in these languages remains an activity that has be specified without referring to run-time information, such that aspects are deployed on all hosts where they may potentially apply. If not, aspects may not apply when expected.

2.2 Dynamic Deployment of Aspects

Dynamic deployment of aspects addresses the issues of static deployment by avoiding the cluttering of pointcut definitions with cumbersome dynamic conditions. It augments the reuse potential of aspects by allowing certain scoping decisions to be deferred to aspect deployment time.

Dynamic deployment of aspects is usually found in more expressive aspect languages where aspects are (at least to some extent) first-class entities. For instance, in CaesarJ, aspects are just objects that happen to have some pointcuts as attributes. A language like AspectScheme [6] goes further, since both pointcuts and advices are first-class functions, enjoying the full power of higher-order programming patterns. This is in sharp contrast with languages like AspectJ and AWED where aspects are mostly first-order entities.

Some approaches offer dynamic deployment with global scope [1, 6], however the semantics of this mechanism in presence of multi-threaded programs is unclear. In contrast, several *structured* dynamic deployment mechanisms have been provided, with clearer semantics. For instance, both CaesarJ and AspectScheme support a dynamically-scoped (thread local) deployment construct, like `deploy(asp){`*`block`*`}`, whereby the aspect instance `asp` sees any join point produced in the dynamic extent of the execution of *`block`*. AspectScheme also supports statically-scoped deployment, in which the aspect instance `asp` sees any join point produced *lexically* in the body of `block` (including in future applications of functions that may escape the block). This resembles per-object deployment [1, 14], like `deploy-on(obj,asp)`, whereby `asp` sees any join point that occurs in the context of the object `obj`.

To unify and subsume all these variants of scoping semantics for dynamically-deployed aspects, Tanter has proposed *deployment strategies* [16]. A deployment strategy specifies the scoping of an aspect through three components: how it should propagate on the call stack (dynamic scoping dimension), how it should propagate along created procedural values (static scoping dimension), and if the pointcuts of the aspect should be refined locally for a given deployment. These components are themselves pointcuts. For instance, consider the following (artificial) banking example: We wish to use a general-purpose logging aspect `log` to log all modifications of the list of lenders on an open loan. Such modifications only happen in calls to `Loan` objects with a `Client` parameter. Also, this will never happen beyond the database facade `DBAccess`. The following code deploys `log` over execution of *`block`*:

```
deploy[!target(DBAccess),target(Loan),
       if(argOfType(Client))](log){ block }
```

The deployment strategy specifies that *(a)* the aspect sees all join points in the dynamic extent of the block except when the target of

[1]Technically, in Java RMI, this depends on whether or not the affected class declares a consistent `classVersionUID` field in both hosts. If so, the aspect does not apply. Otherwise, classes from both hosts are considered incompatible [15].

```
class TravelService {
   ...
  Booking bookPackage(FlightSpec fsp, RoomSpec rsp,
                      Traveler trv, Selector sel){
    FlightsInfo ft = DBAccess.current.reserve(trv,fsp);
    RoomsInfo rm = hotelService.reserve(trv,rsp);
    Reservation res = sel.pick(ft,rm);
    return confirm(res);
  }
  Booking confirm(Reservation res){
   ... confirm the reservations ...
   ... obtain weather information ...
   ... complete travel info and return ...
  } }
```

Figure 1: Section of the travel service: booking of a package

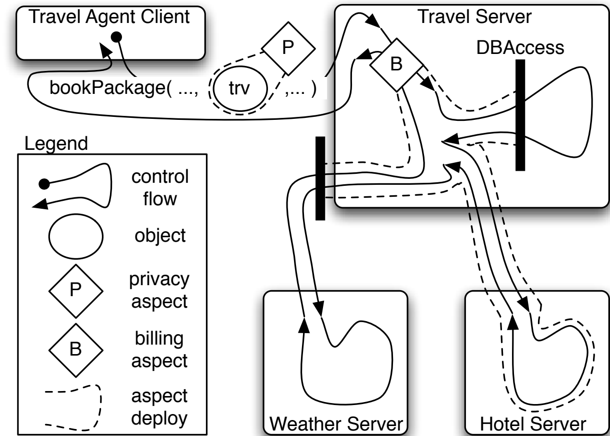

Figure 2: Travel agent system, focusing on the `confirm` phase. Privacy and billing aspects deployed

a call is of type `DBAccess`; *(b)* the aspect is captured in all created `Loan` objects so that it will see join points produced in their context, even if they happen outside of the dynamic extent of the block); *(c)* logging is refined to apply only if a join point has an argument of type `Client`. Deployment strategies subsume existing proposals and enhance aspect reusability by giving fine-grained control over the scope of dynamically-deployed aspects [16].

2.3 Dynamic Distributed Aspect Deployment

CaesarJ is the only aspect language with dynamic deployment that supports distribution to some extent. Beyond global deployment, CaesarJ supports a structured form of dynamic deployment, on a distributed control flow. The advantage of this solution over static deployment approaches is that the aspect is automatically deployed on the control flow in remote hosts as needed. This avoids the different problems mentioned in Section 2.1.

Distributed per-thread deployment is however but one point in the design space of distributed aspect scoping semantics (per-this deployment in CaesarJ only works locally). Conversely, deployment strategies cover that space, but are formulated in a non-distributed context. This paper therefore explores support for expressive scoping of distributed aspects, by proposing means to augment the power of deployment strategies to express distribution-related constraints on the scope of deployed aspects.

3. EXPRESSIVE SCOPING SCENARIOS

We now present several distributed scenarios for which existing languages provide insufficient deployment support. We informally describe a number of deployment strategies that solve these scenarios. Section 4 then overviews our proposal and concisely expresses these desired strategies. Recall that the focus of this work is to provide expressive scoping through the use of deployment strategies. The modification of pointcuts by adding extra conditions to yield similar scoping results is considered a poor substitute, as discussed in Section 2.1 and [16]. So we wish to avoid such modifications.

Running Example. As a running example we consider a typical client-server system for travel agents. Travel agents use the client application to book travel packages that include a flight and hotel reservation. The travel server application handles flight booking locally, but delegates hotel reservations to a secondary booking server of another company. As a courtesy, the typical climate conditions for the destination at that time are also supplied to the traveler. For this a free weather service is used.

To book a package, the travel agent specifies constraints on flights and hotels, and sends a request to the travel server. The travel server gathers candidate reservations, of which a combination is selected by letting the travel agent pick items from a list. An outline of the travel server code for this scenario is shown in Figure 1. The `trv` and `sel` parameters of the `bookPackage` service respectively contain relevant information of the traveler, and the object used as a callback to select reservations. Figure 2 illustrates the reservation confirmation phase, with deployment of a billing aspect.

In this system, we want to implement a `Billing` aspect, using an implementation of the wormhole pattern, to avoid cluttering the parameter list of the different functions involved with an extra billing parameter. The implementation of this aspect is shown below:

```
aspect Billing {
 Bill bill = new Bill();
 pointcut billMe(TravelInfo inf):
    execution(* *.resConfirmed(TravelInfo)) && args(inf);
 after(TravelInfo inf) returning: billMe(inf) {
      bill.addItem(inf); } }
```

When confirming a reservation, the travel server as well as the booking server first verify if the candidate reservations are still valid (e.g. they have not expired). They then call a `resConfirmed` method on themselves, the argument type of which is a superclass of `FlightsInfo` and `RoomsInfo`, to update their internal bookkeeping. The name and parameter list of this method is fixed by convention, which allows the `Billing` aspect to apply.

An implementation of the `Billing` aspect using AWED requires changing the `billMe` pointcut as follows:

```
pointcut billMe(TravelInfo inf):
    execution(* *.resConfirmed(TravelInfo)) && args(inf)
      && host(TravelGroup)                              [1]
      && cflow(call(* TravelService.confirm(..)));      [2]
```

In this pointcut, in [1], `TravelGroup` is a group of hosts, explicitly defined as `{travelServer, hotelServer}`. This means that the distribution setup is embedded in the pointcut definition. Also, the pointcut must explicitly define the control flow of interest in [2]. Reusing `Billing` in a different context or with a different deployment strategy would require yet another rewrite of the pointcut.

A better solution is to use CaesarJ as it provides for dynamic deployment on a distributed control flow. An example of this is changing the `confirm` call to the code below[2]:

[2]CaesarJ actually provides an API rather than specific syntax for this type of deployment [1]. For consistency with the rest of this paper we assume the existence of some syntactic sugar for these API calls.

```
deploy-distcflow(new Billing()){ return confirm(res); }
```

The example shows how dynamic deployment enables reuse of an aspect as the pointcut of the `Billing` aspect remains unchanged.

3.1 Case 1: Controlling propagation

Let us refine the example, to address two issues: *a)* the aspect propagates to the free weather service, where it will never apply; *b)* in the travel server we know that behind the database facade `DBAccess` no `resConfirmed` call will ever be made, so we should not propagate the `Billing` aspect beyond the facade. We therefore want to change the previous deployment to a specific deployment strategy, say, `deploy-TravelGroup-cflow-except-DBAccess` that cuts propagation at these points. This would be achieved by changing the call to `confirm` as below:

```
deploy-TravelGroup-cflow-except-DBAccess(new Billing()){
  return confirm(res);
}
```

However, CaesarJ does not provide for such a deployment strategy as there is no way to control propagation based on a target host or receiver type. AWED has the advantage that it does provide control of propagation on hosts (the `TravelGroup` in the example), however it does not allow for specification of filtering out propagation on receiver type, and has static deployment.

3.2 Case 2: Controlling activation

Suppose that performance information needs to be gathered from the client. We wish to reuse an existing `Profiling` aspect (that uses a generic `execution(* *.*(..))` pointcut). This requires the aspect to be deployed such that the entire control flow of the package booking front-end is captured. However we must exclude the computation of the server, invoked through the `bookPackage` remote method. This yields the following deployment:

```
deploy-active-locally(new Profiling()){ booking code };
```

The `deploy-active-locally` strategy is not equal to simply stopping propagation at the host boundary. This is because there is a callback from the server to the `Selector` object given as parameter to the `bookPackage` method. Recall that this callback pops up a dialog box that allows the travel agent to pick among proposed reservations for a given package. We also want to gather profiling information for these dialog boxes.

Again, this deployment strategy cannot be expressed in current distributed aspect-oriented languages. CaesarJ's dynamic deployment on a distributed control flow does not allow specification of deactivation on given hosts. We can achieve similar scoping with AWED, but with a reuse cost as the pointcut must be modified.

3.3 Case 3: Controlling per-object activation

As a last example, consider the `Traveler` object passed to the travel service server. This object contains all the traveler information the client has, including e.g. address and phone number. For privacy reasons the client must not reveal such information to hosts outside of the `TravelGroup`. The interface to the different servers however must not be changed (*e.g.* to use a new restricted interface), so this feature is implemented using a `Privacy` aspect as follows:

```
aspect Privacy {
   pointcut protectMe() :
     execution(String *.getAddress()) ||
     execution(String *.getPhone()) || ... ;
   String around() : protectMe(){ return "N/A"; }}
```

The aspect overrides selected getter functions to return `"N/A"`. In this scenario, we want the `Privacy` aspect to be embedded in all `Traveler` objects, which happen to be obtained from a factory:

```
TravelerFactory fact =  ...
Privacy priv = new Privacy();
deploy-in-Traveler-inactive-TravelGroup(priv, fact);
```

The different deployment syntax is due to such embedding being different from deployment on execution, as seen in the previous two cases. Here, similarly to statically-scoped aspects in AspectScheme, the aspect is deployed in the factory and propagates on object creation, with the restriction that the object being created is of class `Traveler`. Being embedded in `Traveler` objects created by the client, the aspect follows these objects as they are sent over the network, as depicted on Figure 2. In addition, the deployment strategy specifies that the aspect is only active on (untrusted) hosts that are not part of the `TravelGroup`.

It is impossible to achieve this scoping semantics using current proposals that support per-object aspects, AspectJ and CaesarJ, because these only work locally. Furthermore, even if it would work in a distributed setting, deploying the aspect on the factory will not further propagate it on objects created by the factory. While this could be addressed using an auxiliary deployment aspect, we would still be unable to control activation based on host properties.

4. SCOPING OF DISTRIBUTED ASPECTS

The previous examples illustrate that relevant scenarios of dynamic distributed aspect deployment are not well-served by state-of-the-art distributed aspect languages and frameworks. After an overview of plain deployment strategies and a brief analysis of what is missing in a distributed setting, this section sketches our proposal for expressive scoping of distributed aspects, and shows how we can express the previous cases succinctly. A detailed description of the semantics of our proposal is deferred to Sections 5 and 6.

4.1 Plain Deployment Strategies

(Non-distributed) deployment strategies, also called here plain strategies in order to avoid possible ambiguities, have been introduced by Tanter as a particularly expressive mechanism to specify the scoping of dynamically-deployed aspects [16]. In brief, a deployment strategy is a specification of the form $\delta\langle c, d, f\rangle$, where c and d are *propagation functions* used to specify, respectively, call stack propagation and propagation within delayed evaluation (created functions or objects), and f is a *join point filter* used to express deployment-specific filtering of the join points seen by the deployed aspect. All three components are pointcuts, *i.e.* they take a join point as parameter and return a boolean.

Dynamic aspect deployment is then performed with a deployment expression: $deploy(a, \delta\langle c, d, f\rangle, e)$, where e is a reducible expression. Aspect a is deployed *during the evaluation of* e, with propagation strategy δ. Additionally, in this paper we also use the ability to deploy an aspect on an existing procedural value using $deploy\text{-}on(a, \delta\langle c, d, f\rangle, e)$. In this case, e is first reduced to a procedural value. Aspect a is deployed *within that value*, with propagation strategy δ. In a functional (resp. object-oriented) setting, this means that a is deployed over the evaluation of the function body (resp. method bodies), each time the function (resp. one of the methods) is applied[3].

[3] As discussed in [16], *deploy-on* is more powerful than per-instance aspect deployment as found in CaesarJ, composition filters [4], and AspectJ's per-this aspects, because in these proposals, aspects do not propagate (*i.e.* they are deployed on an object with deployment strategy $\delta\langle false, false, true\rangle$).

4.2 Analysis of the Problem

The requirements of the distributed deployment scenarios of Section 3 suggest that, in addition to specifying a deployment strategy for the dynamic deployment of an aspect, one needs to be able to specify the following properties related to distribution:

- *Locality of aspect propagation*: firstly, when an aspect is propagated along the call stack [Case 1], should it be propagated when a remote call is performed? secondly, when an aspect is attached to a procedural value (function, object) that is passed by copy to a remote host [Case 3], should it remain attached to the copy of that value?
- *Locality of aspect activation*: whenever an aspect is dynamically deployed and propagated to various hosts [Case 2], on what host should it be active?
- *User-defined notions of locality*: it should be possible to express locality of propagation and activation of a dynamically-deployed aspect based on any criteria related to the *properties* of the considered hosts, in order to go beyond the local-remote-global trichotomy [Cases 1 & 3].

4.3 Distributed Deployment Strategies

How shall we add the above-mentioned properties to plain deployment strategies? The basic idea is to define distributed deployment strategies as transformers over plain deployment strategies: an application $t(\delta)$ of a strategy transformer t to a (non-distributed) deployment strategy δ augments the latter by a specification of distributed propagation and/or activation. In this way, distributed strategies permit to abstract over the two propagation dimensions (c for call stack and d for delayed evaluation) and provide a single distributed propagation mechanism.

The previous analysis directly suggests the introduction of the following six simple distributed deployment strategies:

1. $propagate\text{-}global$: always propagate.
2. $propagate\text{-}local$: never propagate to other hosts.
3. $propagate\text{-}remote$: only propagate in remote calls.
4. $active\text{-}global$: always active.
5. $active\text{-}local$: never active in other hosts.
6. $active\text{-}remote$: active only in remote hosts.

By default, we consider that an aspect propagates and is active on all hosts (*i.e.* 1 and 4 are the default). The programmer only needs to override this default.

The simple distributed deployment strategies can in turn be defined in terms of two general distributed strategies $propagate\text{-}if$ and $active\text{-}if$ that are parametrized by a host predicate hp. Such a predicate defines a subset of all hosts based on a host representation, thereby allowing for user-defined notions of locality:

- $propagate\text{-}if(hp)$: propagate to hosts matched by hp.
- $active\text{-}if(hp)$: active on hosts matched by hp.

4.4 Expressing the Examples

Syntax. When illustrating our model with a Java-like language, we use the following variation on the CaesarJ `deploy` syntax:

```
deploy ::= deploy[ds](asp){ expr }
deploy-on ::= deploy-on[ds](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.

Since distributed deployment strategies are functions that take other strategies as parameters, we assume strategies to be first-class values in the language. We introduce a bracketed syntax `<...>` to construct deployment strategies as values. For instance, `dynamic = <true, false, true>` defines the dynamically-scoped deployment strategy found in both CaesarJ and AspectScheme (call stack propagation only, no filter, see 4.1).

Case 1. The billing aspect is deployed over a dynamic extent, bound by a condition on the receiver type in order to avoid propagating beyond the `DBAccess` facade. This bounded dynamic scope is expressed by a plain deployment strategy as:

```
notBeyondDB = <!target(DBAccess), false, true>;
```

Here, `target` is similar to the AspectJ`target` pointcut designator, selecting join points based on the type of the target object. If we were to deploy the aspect on all hosts, we could use the $propagate\text{-}global$ strategy:

```
deploy[propagate-global(notBeyondDB)](billing){...}
```

Note that this `propagate-global` specification is optional because it corresponds to the chosen default. In any case, the scenario stipulates that the aspect should only propagate in hosts that belong to the `TravelGroup`. This is done as follows:

```
deploy[propagate-if(inTravelGroup)(notBeyondDB)]
  (billing){...}
```

where `inTravelGroup` is the appropriate host predicate.

Case 2. In this case, the profiling aspect should be propagated globally (the default), but only *active* locally. This corresponds to $active\text{-}local$:

```
deploy-on[active-local(dynamic)](prof, client);
```

where `dynamic` refers to dynamically-scoped deployment like in AspectScheme and CaesarJ, as defined above.

Case 3. In this case, without considering distribution, the deployment strategy is a refinement of statically-scoped deployment *a la* AspectScheme, where the aspect is captured in all objects created by the factory that are of type `Traveler`. With respect to distribution, the protection aspect should only be active in objects accessed on remote hosts. As a first step, let us assume that this is on any remote host:

```
inTravelers = <false, target(Traveler), true>;
deploy-on[active-remote(inTravelers)](priv, fact);
```

Note that we use `deploy-on` to deploy the aspect in the factory object. Actually, the scenario stipulates that the aspect is only active in hosts that are not part of `TravelGroup`:

```
deploy-on[active-if(not(inTravelGroup))(inTravelers)]
  (priv, fact);
```

To summarize, all the examples can be expressed by augmenting the plain deployment strategies with the ability to propagate and activate aspects depending on the host and rely on typical `deploy` constructs to apply these strategies. The remainder of this paper details the semantics of our proposal, using a progressively-extended Scheme interpreter.

```
(define (eval exp E)
 (cond
  ((lit? exp) (lit-value exp))
  ((var? exp) (env-lookup (var-name exp) E))
  ((set? exp) (env-set! (set-name exp)
                  (eval (set-nval-exp exp) E) E))
  ((lambda? exp) (make-closure (lambda-params exp)
                               (lambda-body exp) E))
  ((app? exp) (let ((cl   (eval (app-fun exp) E))
                    (args (eval-args (app-args exp) E)))
                (eval (closure-body cl)
                      (extend-env (closure-params cl)
                                  args
                                  (closure-env cl)))))
  ...))
```

Figure 3: Interpretation of a higher-order procedural language.

5. A MODEL OF DISTRIBUTED ASPECTS

This section gradually introduces a model of distributed AOP that covers the relevant parts of the current state of the art in distributed aspect languages. We start by introducing a small Scheme-like higher-order procedural language, add distribution to it, then aspects, and finally add some typical distributed aspect support. This model is further extended in Section 6 to fully support distributed deployment strategies as sketched in Section 4.

5.1 Core base language

We start with a higher-order procedural language with literals (numbers, strings, booleans), variables with mutation and first-class functions with call-by-value. In addition, the language supports a set of typical primitives absorbed from Scheme itself. The only data structure supported are cons cells, and by extension, lists (all introduced as primitives as well).

We deliberately do not include objects in the model, both for space reasons and because our descriptions can be transposed to objects fairly directly. The essence of this transposition can be found in the original work on deployment strategies, in which Tanter provides definitional interpreters and semantics for aspect languages for both functional and object-oriented base languages [16].

Interpretation. We give the operational semantics of our language using definitional interpreters[4] written in environment-passing style [7]: 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 `lit?`, and accessed with accessors such as `lit-value`.

Figure 3 describes the interpreter for the base language. Accessing and setting a variable is done by respectively looking up in and mutating the current lexical environment. Defining a function creates a closure that captures its lexical environment. Applying a function evaluates the body of the closure in its definition-time environment extended with new bindings for the formal parameters.

5.2 Adding distribution

We now extend the language with support for distribution. We introduce a means to export functions so that they can be referenced and applied from a remote host, via a function stub. Like in standard remote procedure call and remote method invocation, remote function application is synchronous.

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

```
(define (eval exp E)
  (cond ...
        ((app? exp)
         (let ((f    (eval (app-fun exp) E))
               (args (eval-args (app-args exp) E)))
           (eval-app f args))))
        ...))

(define (eval-app f args)
  (if (fun-stub? f)
      (remote-exec f args)
      (eval-exec f args)))

(define (eval-exec f args)
  (eval (closure-body f)
        (extend-env (closure-params cl)
                    args
                    (closure-env cl))))

(define (remote-exec f args)
  ...serialize host, function id & arguments...
  ...send to target host (triggers receive-call)
  ...wait for result...
  ...deserialize result...)

(define (receive-call)
  ...deserialize client host, function id & arguments...
 (let ((f (lookup-exported id)))
   (eval-exec f args)
   ...serialize result...
   ...send back to caller...))
```

Figure 4: Interpretation of a higher-order procedural language with distribution.

A host runs a local registry of the functions it exports. The programmer can export a function on the current host using the `export` primitive, which returns a stub to that function. Passing this stub as a parameter of remote calls permits other hosts to apply the function remotely. Additionally, one can export a function giving it a name using `export-as`. A remote host can then do a `lookup` of that name on that host in order to obtain a stub to that function.

A stub is a structure that contains the name of the function, the identifier of the host that provides the function, as well as any interesting metadata on the function (such as expected number of arguments or any other type information). Parameter passing is done by copy, but of course, passing a stub by copy is equivalent to passing an exported function by (remote) reference.

Passing a closure (not a stub) by copy implies also copying the transitive closure of its captured environment. In order to avoid copying the full local environment of each host, each host has a global root environment that is not captured by copy and hence never passed over the network. Examples of values that reside in the root environment are typical library and utility functions.

To illustrate, consider the code below run on the booking server, which exports a reservation service under the name “reserve”:

```
(export-as "reserve" (lambda (tinfo specs) ...))
```

On a client, the service can then be looked up and applied:

```
(let ((reserve (lookup "booking server" "reserve")))
  (reserve ... ...))
```

Interpretation. Figure 4 sketches the evolution of the interpreter to support distribution. The interpretation of a function application now needs to discriminate between local and remote calls (`eval-app`). Local calls are interpreted as before. Remote calls

```
(define (eval exp E jp)
  (cond ...
        ((app? exp)
         (let* ((f    (eval (app-fun exp) E jp))
                (args (eval-args (app-args exp) E jp)))
                (njp  (make-jp f args jp)))
           (weave njp)
           (eval-app f args njp))
        ...))
```

Figure 5: Interpretation of a higher-order procedural language with aspects (call join points, before advice).

imply extracting information from the stub, serializing it and sending it to the target host (`remote-exec`). Since remote invocation is synchronous, the interpreter then waits for the result, and deserializes it when available. On the server side, when a call to an exported function is received (`receive-call`), the actual closure is looked up based on its exported name, and evaluated locally. The result is then serialized and sent back to the caller.

5.3 Aspects

We start with a model of join points, pointcuts and advices which is similar to that of AspectScheme (a formal semantics of which can be found in [6]). This model will be extended in Section 6 to support distributed deployment strategies.

The only join points considered for now are function applications. A join point can be either top-level or nested within other active function applications.

A *join point in context* is an abstraction of the call stack: it is represented by a recursive structure, whose head is the current join point (the function to apply), and whose tail is the context (the pending active function applications).

An important characteristic of the model is that both pointcuts and advices are *first-class values*. A pointcut is a predicate over join points in context, *i.e.* it is a function of type[5]:

$$\mathcal{PC} = JoinPoint \rightarrow Bool$$

A pointcut designator, such as `call` and `cflow`, is a function that returns a pointcut. Figure 6 shows how the typical pointcut designators and their composition are defined in the language. For instance, if `reserve` is a function in scope, then `(call reserve)` returns a pointcut that matches application of that function.

Similar to our previous work on aspect scoping [16], 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 where advices are modeled as function transformers, we simply model advices as functions of type:

$$\mathcal{ADV} = JoinPoint \rightarrow Unit$$

an advice performs its effect before the standard interpretation proceeds, and its return value is ignored. We do not account for context exposure beyond the fact that an advice receives the matched join point in context as parameter. Finally, an aspect $a \in \mathcal{ASP}$ is simply represented as a pair of a pointcut and an advice.

[5]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.

```
(define call (lambda (f)
               (lambda (jp) (eq? (jp-fun jp) f)))))

(define cflow (lambda (pc)
                (lambda (jp) (or (pc jp)
                                 ((cflowbelow pc) jp)))))

(define cflowbelow (lambda (pc)
        (lambda (jp) (and (has-parent? jp)
                          ((cflow pc) (jp-parent jp))))))

(define && (lambda (pc1 pc2)
             (lambda (jp) (and (pc1 jp) (pc2 jp)))))
```

Figure 6: Some typical pointcut designators.

Interpretation. To model join points in context, the interpreter takes as parameter the join point at the enclosing function application. When a function is to be applied, the interpreter creates a new join point representing that application, triggers weaving, and then proceeds with executing the function application, with the new join point. This is outlined in Figure 5 (to be compared to the initial definition of `eval` in Figure 3).

A join point is a structure that aggregates the applied function, the arguments, and its parent join point (`#f` at the root):

```
(define-struct jp (fun args parent))
```

At this stage, we simply consider a global aspect environment, *i.e.* a global variable in the interpreter. Weaving simply iterates over all the aspects in this global environment, applying their pointcuts to the new join point, and applying the associated advice whenever a pointcut matches[6].

5.4 Distributed Aspects

Deployment. Current distributed aspect languages and frameworks provide different mechanisms to deploy aspects on hosts in a network. Except for CaesarJ's distributed control flow deployment, these specifications are all static, and imply that aspects have global scope on each host.

As a first step, we introduce a mechanism to deploy an aspect on a given (set of) host(s). Like existing proposals, this mechanism gives aspect a global scope on each host; however, in our model aspects are not statically-specified entities. Therefore, our per-host deployment mechanism is dynamic. Executing:

```
(let ((pc  (call reserve))
      (adv (lambda (jp) ...))
 (deploy-global-on-host "host1" (pc . adv))))
```

deploys an aspect that operates on calls to `reserve` on host `host1`. The implementation adds the aspect to the global aspect environment of that host. This global deployment scheme is refined in the next section when introducing distributed deployment strategies.

Remote pointcut and advice evaluation. When aspects are passed over the network, for instance when they are deployed to a remote host, like above, they are treated as plain values, passed by copy.

[6]In AspectScheme, the execution of pointcuts and advice triggers further join points, so a dedicated primitive is provided to be able to define pointcuts and avoid these applications to trigger join points. In our case, for simplicity, we execute pointcuts and advices in "sandboxes" where no aspect can match, thereby avoiding other aspect reentrancy issues. For a general discussion about reentrancy issues with aspects, in particular with first-class pointcuts and advices, we refer the reader to [17].

But since pointcuts and advices are first-class functions, they can also be *remote* functions (*i.e.* stubs): in that case, they are passed by reference over the network and are always executed on their exporting host.

This difference matters when one considers that these functions can be stateful: they encapsulate their lexical environment, which can be mutated. For instance, in the deployment example above, suppose that `adv` is a stateful function, like a counter. `adv` is being passed by copy, so on `host1`, `adv` will have its own local state. On the contrary, if one would deploy it as:

```
(deploy-global-on-host "host1" (pc . (export adv)))
```

the advice function is passed by remote reference, since it is first exported using `export`. Therefore its state does not reside on `host1` but on the host on which the deployment is performed.

Join points and distribution. Because a join point in context keeps a parent link to its predecessor join point, it is an abstraction of the call stack (Section 5.3). The interpreter always passes the current join point around, including upon remote calls. Therefore, the stack abstraction is maintained upon distribution, resulting in a representation of a distributed call stack.

Performance-wise, passing this stack abstraction by copy upon each remote call is not appropriate. Better solutions can be devised. For instance, the join points can be stored locally by default and lazily copied, with the exception of a few join points at the top of the stack. However, as this has no influence on the semantics of aspect deployment and execution, we will stick here to the simple model presented in the previous paragraph.

Distribution-related pointcuts. With respect to expressiveness of the aspect language, it has been repeatedly shown that being able to discriminate join points based on their host of occurrence is valuable [3, 13, 18], *e.g.* to express pointcuts that match only on certain hosts. To this end, we extend the representation of a join point to embed its host of occurrence. We also go a step further by including the target host of a call. The target host of a call join point is only specified if the call is a remote call. This provides the ability to discriminate join points not only based on where they occur, but also if they are remote calls or not, and to which host they are directed. In order to be able to reason about hosts, we represent hosts by a set of key-value properties, as in ReflexD [18]. This includes the possibility to define host groups like in AWED [3].

Figure 7 presents a number of pointcuts and pointcut designators that take advantage of these extensions. For instance, `host` matches a join point only if its host matches the given set of properties. `remote-call?` discriminates remote calls from local ones. We can also define local-only versions of the control flow pointcut designators, so as to ensure that pointcut matching does not involve inspecting the remote stack.

5.5 Extending the Execution Model

While the model of distributed aspects presented up to here is fairly complete, we extend it further with two refinements. These refinements –namely, the separation of call and execution, and of definition and copy of functions– are *not specific* to distribution. However, they are natural in a distributed setting. In addition, when combined with the basic distributed AOP features we have already presented, they make it possible to define distributed deployment strategies as simple transformers of plain strategies, as will be shown in Section 6.

Call and execution. The first refinement is to split "function application" into function call and execution. Most non-distributed

```
(define host
   (lambda (props)
     (lambda (jp) (host-match? props (jp-host jp)))))

(define local-cflow
  (lambda (pc)
    (lambda (jp) (or (pc jp)
                     ((local-cflowbelow pc) jp)))))

(define local-cflowbelow
  (lambda (pc)
    (lambda (jp)
      (and (has-parent? jp)
           (same-host? jp (jp-parent jp))
           ((local-cflow pc) (jp-parent jp))))))

(define remote?
  (lambda (jp) (not (eq? (jp-target-host jp) #f)))

(define remote-call?
  (lambda (jp) (and (call? jp) (remote? jp))))

(define remote-copy?
  (lambda (jp) (and (copy? jp) (remote? jp))))
```

Figure 7: Distribution-related pointcut designators.

aspect languages actually make this distinction. In a distributed setting, it makes even more sense because both join points potentially happen on different hosts: *e.g.* the call on the client host, and the execution on the server. So, we introduce a new kind of join point to denote function execution. Such a join point is created on the host that evaluates the actual body of a function.

Since up to now we had only one kind of join points, we need to refine the definition of join points with an extra kind attribute. To sum up, a join point is now defined as:

```
(define-struct jp (kind fun args parent host target-host))
```

where (up to now) `kind` can be either `call` or `exec`.

Creation and copy. The second refinement is to consider that a function can not only be created when defined in program text, but also whenever a function is copied[7]. In a distributed setting, this is important because arguments to remote functions are passed *by copy*. We introduce two new kinds of join points, `new` and `copy`, to denote "fresh" function creation and function copy, respectively. A `copy` join point holds the original function in its `fun` attribute.

Just introducing a copy operation in the execution model enables us to talk about remote parameter passing uniformly, in the same way as we are able to discriminate remote calls. Figure 7 shows the definition of the `remote-copy?` pointcut.

6. DYNAMIC ASPECT DEPLOYMENT

Section 5 has introduced a simple model of distributed aspects, which covers a good part of the features of current distributed aspect languages/frameworks like remote pointcuts [13], AWED [3] and ReflexD [18]. The model includes four kinds of join points (call, execution, new, copy), potential remote evaluation of both pointcuts and advices, property-based representation of hosts, and embedding of current and target hosts in join points to support pointcuts that can discriminate local and remote calls, as well as the host of occurrence.

The deployment model so far supports dynamic per-host deployment of aspects with host-global scope. As illustrated in Section 2

[7]Note that introducing copying into the model brings us even closer to objects, specifically regarding cloning and instantiation.

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

Figure 8: Deployment strategies semantics in a nutshell

and 3, explicit per-host deployment of aspects with host-global scope does not suffice (even if being able to do it dynamically is already a gain over static approaches). In this section, we refine this model to support expressive scoping of dynamically-deployed aspects in a distributed context.

We quickly review the semantics of plain deployment strategies as presented by Tanter [16]. Then, we show how to extend these semantics to take into account the extended execution model with the join points introduced in Section 5.5, and describe an definitional interpreter of our proposal. Finally, we define distributed deployment strategies as transformers of deployment strategies, expressing the solutions to the examples given in Section 4.4.

6.1 Background: Deployment Strategies

Aspect deployment strategies have initially been proposed in a non-distributed context [16], their semantics being defined operationally using Scheme interpreters in the line of the Aspect SandBox project [10, 11, 19].

Compared to the environment-passing style interpreters we have presented until now, an interpreter of dynamic deployment strategies evaluates an expression within an *aspect environment* that is passed around between evaluation steps[8].

The aspect environment contains the currently-deployed aspects whose pointcuts must be evaluated. An aspect is initially inserted into the environment when it is deployed, with its strategy. The expression $deploy(\delta\langle c, d, f\rangle, a, e)$ inserts $\langle a, \delta\langle c, d, f\rangle\rangle$ in the current aspect environment A before proceeding with the evaluation of the reducible expression e. To support statically-scoped aspects, a closure also captures an aspect environment ($closure.A$). It is possible to augment this captured aspect environment using $deploy\text{-}on(\delta\langle c, d, f\rangle, a, v)$, which inserts $\langle a, \delta\langle c, d, f\rangle\rangle$ in the aspect environment of the procedural value v[9].

Figure 8 recalls the semantics of deployment strategies in a non-distributed context. The aspect environment A is a set of pairs $\langle a, \delta\langle c, d, f\rangle\rangle$, where a is an aspect and c, d, and f are the components of the deployment strategy δ.

When a function is defined, the corresponding closure captures only those aspects whose propagation function for delayed evaluation d returns $true$. This forms the set A_{def}.

When a function is applied, its body is evaluated in an aspect environment comprised of the aspects in the current aspect environment whose propagation function for call stack c returns $true$, in addition to the aspects in the aspect environment of the closure. This forms the set A_{app}.

The set of aspects A_{weave} that should be woven at a given join point is obtained by selecting the aspects of the current aspect environment whose join point filter f accepts the current join point.

[8] The Aspect SandBox interpreter of Masuhara *et al.* [11] and the formal model of Wand *et al.* [20] use a global aspect environment: this is insufficient for modeling expressive aspect scoping [6, 16].

[9] The $deploy$ expression can actually be seen as syntactic sugar for $deploy\text{-}on$, which transforms the expression into a value by nesting it under a lambda: $deploy(\delta, a, e) \rightarrow deploy\text{-}on(\delta, a, \lambda().e)$

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

Figure 9: Revisiting deployment strategies semantics.

6.2 Refining Deployment Strategies

Our analysis of Section 4 has made clear that we need to be able to express the locality of aspect propagation and activation. We have found that we can bring this extra expressiveness to deployment strategies by introducing the two refinements to the execution model introduced in Section 5.5. It is sufficient to discriminate between call and execution join points, and add support for copy join points to the model. The original exposition of deployment strategies [16] however only considers function application join points. Therefore the semantics of deployment strategies as presented in Figure 8 needs to be revisited.

First, to account for the separation of call and execution join points, we update the semantics of deployment strategies to separate A_{app} into two aspect environments: A_{call}, the set of aspects that propagate on the call stack, and A_{exec}, the set of aspects to use during the evaluation of the body of a function:

$$
\begin{aligned}
A_{call} &= \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid c(njp)\} \\
A_{exec} &= A_{call} \cup closure.A
\end{aligned}
$$

Note that A_{call} is computed on the caller side, so if an aspect does not propagate on remote calls, it is not sent over the network.

Second, the logic of delayed evaluation propagation, d, needs to be updated to take into account function copying. We rename A_{def} to A_{new} and distinguish a new aspect environment, A_{copy}, which is the set of aspects that are embedded in a function copy:

$$
\begin{aligned}
A_{new} &= \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \mid d(njp)\} \\
A_{copy} &= \{\langle a, \delta\langle c, d, f\rangle\rangle \in A \cup original.A \mid d(njp)\}
\end{aligned}
$$

When a function is copied, aspects captured in the original function ($original.A$) may or may not propagate in the copy: the d function of each aspect deployed in the original function receives the copy join point and decide whether or not the aspect propagates. The other part of the definition of A_{copy} follows the definition of A_{new}: the copied function is a newly-created function, so aspects in the current aspect environment A may be captured: the copy join point is passed to their d propagation function.

Figure 9 summarizes the semantics of deployment strategies with the two refinements mentioned above. (A_{weave} is unchanged.) In the following section we present the corresponding interpreter. Section 6.4 then shows that armed with these new definitions, combined with the definition of join points augmented for distribution (with current and target hosts), distributed deployment strategies can be expressed simply as deployment strategy transformers.

6.3 Interpretation

We now describe the semantics of our updated model of deployment strategies using an interpreter-based operational definition. The interpreter of Figure 10 extends the various interpreters presented in Section 5.

```
(define (eval exp E A jp)      ⟵ [3]
  (cond ...
        ((lambda? exp)  (new-function (lambda-params exp) (lambda-body exp) E A jp))
        ((app? exp)  (let ((f    (eval (app-fun exp) E A jp))
                           (args (eval-args (app-args exp) E A jp)))
                       (eval-call f args A jp))))
        ...))

(define (eval-call f args A jp)
  (let ((njp (make-jp 'call f args jp (get-current-host) (target-host f))))      ⟵ [4]
    (weave-some A njp)      ⟵ [5]
    (let ((asps (collect-match-c njp A)))      ⟵ [6]
      (if (fun-stub? f)
        (remote-exec f args asps jp)
        (eval-exec   f args asps jp)))))

(define (eval-exec cl args A jp)
  (let ((njp  (make-jp 'exec cl args jp (get-current-host) #f)))      ⟵ [7]
        (env  (extend-env (closure-params cl) args (closure-env cl)))
        (asps (union A (closure-aspects cl)))      ⟵ [8]
    (weave-some A njp)      ⟵ [9]
    (eval (closure-body f) env asps njp)))

(define (new-function formals body E A jp)
  (let* ((njp  (make-jp 'new #f formals jp (get-current-host) #f))      ⟵ [10]
         (asps (collect-match-d njp A)))      ⟵ [11]
    (weave-some A njp)      ⟵ [12]
    (make-closure formals body E asps)))

(define (copy-function orig A jp)
  (let* ((njp  (make-jp 'copy orig (closure-params orig) jp (get-current-host) (get-target-host)))      ⟵ [13]
         (asps (union (collect-match-d njp A) (collect-match-d njp (closure-aspects orig)))))      ⟵ [14]
    (weave-some A njp)      ⟵ [15]
    (make-closure (closure-params orig) (closure-body) (deep-copy (closure-env orig)) asps)))

(define (weave-some A jp) (weave (collect-match-f A jp) jp))      ⟵ [16]
```

Figure 10: Interpretation of deployment strategies for a higher-order procedural language, with distribution and extended execution model.

First of all, note that `eval` takes as parameter the current aspect environment, in addition to the lexical environment and the current join point [3]. When a function is applied (`eval-call`), first the corresponding call join point is created [4]. The join point embeds the current host, as well as the target host (`#f` if the function is not a stub). Weaving on the call join point is then triggered [5]: `weave-some` uses `collect-match-f` to obtain A_{weave}, the set of all given aspects for which the join point filter f yields true [16].

Next, all aspects that propagate on the call stack as specified by their c propagation function (*i.e.* A_{call}) are gathered with `collect-match-c` [6]. If the function is a stub, the remote call execution is triggered as explained in Section 5.2, additionally passing the aspect environment A_{call} and the new join point as parameters.

When a function is executed (`eval-exec`), the corresponding execution join point is created [7] and woven [9]. The set of aspects that potentially apply during the evaluation of the function body, A_{exec}, is computed [8] and evaluation proceeds.

Similarly, when a function is defined (`new-function`), the corresponding new join point is created [10] and woven [12]. Since aspect weaving occurs before the actual function is created, there is no `fun` attribute for such a join point (`#f`), and the formal parameters are given as the `args` attribute. The aspect environment of the closure, A_{new}, is obtained using `collect-match-d`, passing it the newly-created join point `njp` [11].

When a function is copied (`copy-function`), a copy join point is created [13]. If the copy occurs during the remote parameter passing process, the join point embeds the target host information. This information is obtained using a dynamically-scoped variable accessor `get-target-host`, used to avoid cluttering all functions with an extra parameter. The join point also embeds the original function. Once the join point is created, the interpreter triggers weaving [15], and creates the closure copy. The closure captures the set of aspects A_{copy}, obtained by using `collect-match-d` with the copy join point on both the current aspect environment, and the aspect environment of the original closure [14].

6.4 Distributed Deployment Strategies

We now show how distributed deployment strategies are expressed as transformers of plain deployment strategies.

Section 4 identified three requirements for distributed deployment strategies: *(a)* locality of aspect propagation, *(b)* locality of aspect activation, and *(c)* user-defined notions of locality. Deployment strategies specify propagation via the call stack c and delayed evaluation d propagation functions; and activation via the join point filter f. By default, an aspect propagates on remote hosts and is active on all hosts, unless its deployment strategy expresses some distribution-related propagation and activation restrictions.

Distributed deployment strategies are transformers that take a deployment strategy and add restrictions to its components. Recall that these components are pointcuts, so the transformations involved are pointcut compositions: combining an existing pointcut with a pointcut that specifies distribution-related conditions.

User-defined notions of locality rely on the possibility to go beyond the local-remote-global trichotomy. The solution to this is to support host properties and arbitrary host predicates (functions from host properties to booleans), as introduced in Section 5.4.

Locality of propagation. Locality of aspect propagation can be obtained by placing restrictions on c and d such that we can control when an aspect is propagated to another host: on a remote call, or when embedded in a function passed by copy. As shown on Figure 7, we are able to provide pointcut descriptors that capture remote calls and copies. So we can express locality of aspect propagation by adding conditions based on the *target host* of the call and copy join points that are used when evaluating the corresponding propagation functions.

More formally, let `restrict-target` be the following higher-order pointcut designator:

```
(define restrict-target
  (lambda (hp) (lambda (pc) (lambda (jp)
    (and (hp (target-host jp)) (pc jp))))))
```

Given a host predicate `hp`, and a pointcut `pc`, `restrict-target` returns a new pointcut, element of $\mathcal{PC}$, which imposes the restrictions of the host predicate `hp` on the target host of the given join point `jp`, in addition to the selection expressed by `pc`.

We can now define the general distributed deployment strategy for propagation, `propagate-if` (Section 4), as follows:

```
(define propagate-if
 (lambda (hp) (lambda (<c, d, f>)
   <((restrict-target hp) c),
    ((restrict-target hp) d), f>)))
```

(For the sake of consiseness, we reuse the bracketed syntax of Section 4.4 to do pattern matching on deployment strategies as well as to build these strategies.) Given a host predicate, `propagate-if` returns a strategy transformer that, given a deployment strategy $\delta\langle \mathtt{c}, \mathtt{d}, \mathtt{f}\rangle$, returns a new strategy where the propagation components `c` and `d` are extended with the host restrictions. The join point filter `f` is untouched.

Locality of activation. Similarly, locality of aspect activation can be obtained by restricting the f component of a deployment strategy. In this case, the conditions are on the *current host* on which the aspect is residing.

Let `restrict-current` be the following higher-order pointcut designator:

```
(define restrict-current
  (lambda (hp) (lambda (pc) (lambda (jp)
    (and (hp (current-host jp)) (pc jp))))))
```

Given a host predicate `hp`, and a pointcut `pc`, `restrict-current` returns a new pointcut, element of $\mathcal{PC}$, which imposes the restrictions of the host predicate `hp` on the current host of the given join point `jp`, in addition to the selection expressed by `pc`.

We can now define the general distributed deployment strategy for activation, `active-if` (Section 4), as follows:

```
(define active-if
 (lambda (hp) (lambda (<c, d, f>)
   <c, d, ((restrict-current hp) f)>)))
```

Given a host predicate, `active-if` returns a strategy transformer that, given a deployment strategy $\delta\langle \mathtt{c}, \mathtt{d}, \mathtt{f}\rangle$, returns a new strategy in which the join point filter `f` is complemented with the host restrictions. The propagation components `c` and `d` are unchanged.

Typical deployment strategies. The six typical deployment strategies introduced in Section 4.3 are easily expressed using the general predicate-based strategies. For instance:

```
(define propagate-local (propagate-if is-current)
(define active-remote   (active-if is-remote)
```

with the following auxiliary functions:

```
(define is-current (eqc? current-host))
(define is-remote  (lambda (h) (not (is-current h))))
```

where `eqc?` is the curried version of `eq?`, and `current-host` is a variable defined on each host. Therefore, `is-current` is a function that compares a given host to its host of origin.

Finer-grained strategies. While distributed deployment strategies are assimilating the two propagation dimensions of deployment strategies into a single one, it remains possible to place different distributed-related restrictions on the individual propagation components. This makes it possible to express, for instance, that an aspect should always propagate on the call stack whatever the host, while it should only propagate in selected hosts when captured in the environment of a procedural value passed by remote copy.

6.5 Discussion

This work introduces a general notion of aspect scoping for distributed programming. Our model includes first-class pointcuts and advices, as well as their potentially remote execution, and exposes four kinds of join points: call and execution, creation, and copy. The latter is rather unusual in mainstream aspect languages, but is essential in order to control by-copy remote parameter passing.

Intentionally, we do not address other crucial issues for distributed aspects, in particular, different underlying communication models. We consider only a simple, purely synchronous model for remote communication, with no particular regard to concurrency. While this corresponds to a wide range of distributed applications, the treatment of concurrency deserves more attention. For instance, in very dynamic distributed contexts like Ambient Intelligence — where dynamic deployment is a highly valuable feature — languages typically adopt an asynchronous communication model, *e.g.* (a variant of) the actor model. Scoping mechanisms need to be refined for properly dealing with these specific assumptions.

This work provides a very expressive model for the propagation of aspects and remote behavior resulting from their activation. This expression power should be counterbalanced by means for the analysis and enforcement of propagation and activation properties, for instance to control aspectual effects and thereby preventing programming errors or enforcing security properties of the resulting distributed applications.

Finally, our model stays at a level of abstraction where function stubs are transparent at the language level. It is possible to devise a lower-level model where stubs are visible and hence deploying aspects on stubs become possible. This may enable the expression, using aspects, of more advanced distribution scenarios like smart proxies, where a stub memoizes results of remote invocations.

7. CONCLUSION

Expressive scoping of dynamically-deployed aspects enhances the potential benefits of aspects in terms of applicability, reuse, and performance by allowing the programmer to defer deployment-related decisions to run-time. Proper aspect scoping is even more crucial in distributed systems so as to avoid inconsistencies due to the decentralized and dynamic nature of these systems. Current aspect languages for distribution however have only very limited support for scoped or dynamic deployment, if any.

In this paper we have discussed the limitations of existing aspect languages in this regard and have provided a number of scenarios that motivate the need for expressive scoping of distributed aspects. To achieve this, we have extended previous work on deployment strategies [16] to deal with the distribution dimension of scoping.

In the line of previous research, as well as the Aspect SandBox project [10, 11, 19], we have given the operational semantics of our proposal as a concise Scheme interpreter.

Distributed deployment strategies provide precise control over the two locality dimensions of aspects in distributed systems: propagation and activation. To achieve this, previous work on deployment strategies is augmented with an extended join point model that, in particular, exposes information about remote calls and copies, as well as about hosts. We then express distributed deployment strategies as transformers of plain deployment strategies. Because deployment strategies are specified dynamically, outside of aspect definitions, aspects can be reused in both non-distributed and different distributed settings.

As a result the solution space for scoping of dynamic deployment of distributed aspects has been explored. Support for such scoping will greatly aid in developing distributed systems using aspects, enabling better reuse of aspects as well as permitting the evolution of such systems. This is especially relevant as this domain is well-known for being particularly subject to crosscutting concerns.

Acknowledgments. We are grateful to the anonymous reviewers for their comments.

8. REFERENCES

[1] 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.

[2] Luis Daniel Benavides Navarro, Rémi Douence, and Mario Südholt. Debugging and testing middleware with aspect-based control-flow and causal patterns. In *In Proceedings of the 9th ACM/IFIP/USENIX International Middleware Conference*, Leuven, Belgium, December 2008. Springer-Verlag.

[3] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 51–62, Bonn, Germany, March 2006. ACM Press.

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

[5] 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.

[6] 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.

[7] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages (2nd ed.)*. The MIT Press, 2001.

[8] 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.

[9] Bert Lagaisse and Wouter Joosen. True and transparent distributed composition of aspect-components. In Maarten van Steen and Michi Henning, editors, *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (Middleware 2006)*, volume 4290 of *Lecture Notes in Computer Science*, pages 42–61, Melbourne, Australia, November 2006. Springer-Verlag.

[10] 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.

[11] 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.

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

[13] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut – a language construct for distributed AOP. In Karl Lieberherr, editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 7–15, Lancaster, UK, March 2004. ACM Press.

[14] 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.

[15] SUN Microsystems. *Remote Method Invocation*, 1998.

[16] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium, April 2008. ACM Press.

[17] Éric Tanter. Controlling aspect reentrancy. *Journal of Universal Computer Science*, 2009. Best Paper Award of the Brazilian Symposium on Programming Languages (SBLP 2008).

[18] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, June 2006. Springer-Verlag. Best Paper Award of the three DisCoTec 2006 Conferences.

[19] 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.

[20] 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.