Science of Computer Programming 75 (2010) 1235–1261

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Scoping strategies for distributed aspects☆

Éric Tanter[a,*], Johan Fabry[a], Rémi Douence[b], Jacques Noyé[b], Mario Südholt[b]

[a] PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile

[b] ASCOLA, École des Mines de Nantes - INRIA, LINA, Nantes, France

ARTICLE INFO

Article history:
Received 7 November 2009
Received in revised form 10 May 2010
Accepted 23 June 2010
Available online 14 July 2010

Keywords:
Aspect-oriented programming
Distribution
Scoping strategies
Dynamic deployment
Scope
Scheme
Operational semantics

ABSTRACT

Dynamic deployment of aspects brings greater flexibility and reuse potential, but requires a 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 scoping 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 and scoping strategies. This work shows that, given some extensions to their original execution model, scoping strategies are directly applicable to the expressive scoping of distributed aspects.

1. Introduction

In the pointcut-advice model of aspect-oriented programming [17,32], as embodied in *e.g.* AspectJ [12], 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 clearly 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 [1,8,18,25], 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

☆ This work is partially funded by the INRIA Associate Team RAPIDS, and the FONDECYT project 1090083 (J. Fabry & É. Tanter). This article extends the article "Expressive Scoping of Distributed Aspects" Tanter et al. (2009) [28] that appeared in the Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD 2009), DOI: http://doi.acm.org/10.1145/1509239.1509245.

* Corresponding author. Tel.: +56 2 978 4953; fax: +56 2 689 5531.
E-mail addresses: etanter@dcc.uchile.cl (É. Tanter), jfabry@dcc.uchile.cl (J. Fabry), douence@mines-nantes.fr (R. Douence), noye@mines-nantes.fr (J. Noyé), sudholt@mines-nantes.fr (M. Südholt).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.06.011

limitations. To tackle these limitations, several distributed aspect languages and frameworks have been proposed [3,19,29]. 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 [9], is therefore required for distributed aspect-oriented programming. Recently, Tanter has proposed *scoping strategies* as a general model for controlling the scope of adaptations in programs [25,26].[1] Applied to aspects, scoping strategies supersede 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 2.4 and show the shortcomings of current AOP languages with distribution support in Section 3.

To address the issue of expressive scoping of distributed aspects, we introduce a set of high-level strategies that specify the locality of aspect propagation and activation, complementing the existing proposal of scoping strategies. Necessary background on this previous proposal is presented in Section 4. As a first step, a sequential model of aspects is built on, as a base language, a prototypical sequential higher-order procedural language. This model is made operational through the definition of a progressively-extended Scheme interpreter. The model is then complemented with scoping strategies. Section 5 extends the sequential model of Section 4 and its definitional implementation with distribution. Section 6 starts by giving an informal presentation of distributed scoping strategies and shows how they concisely express the scenarios considered in Section 2.4. It then dives into scoping strategies in a distributed setting, expounding the semantics of our proposal. Section 7 revisits the scenarios example with our language and discusses why none of the limitations identified with other proposals apply. Section 8 discusses several implications of the work in particular with respect to the weaving approach and its relation to code consistency in a distributed system. Section 9 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:

1. 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 [12], 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.
2. The localization of join points can be explicitly referred to in order to make aspects, in particular pointcut matching, distribution-aware. These are the main characteristics of the models of remote pointcuts [19], Aspects with Explicit Distribution (AWED) [2,3], DyMAC [14], and ReflexD [29]. 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).
3. 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 [1,8,18,25]. 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

[1] In its first presentations [25,28], the model is called *deployment strategy*, but we now find it clearer to separate the notion of *deployment* of an adaptation (which can be realized through different means), from the scoping strategy that specifies the *scope* of the deployed adaptation [26].

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.[2]

The above issue is not the sole issue: such approaches cannot directly express relationships between different distributed entities [19]. 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 [19], 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 [29], 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 to 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.

Some approaches offer dynamic deployment with global scope [1,9], 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 [9] 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 [1,22], 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 *scoping strategies* [25]. A scoping 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) travel booking example: We wish to use a general-purpose logging aspect `log` to log all modifications of dates of an existing reservation. Such modifications only happen in calls to `Reservation` objects with a `Date` parameter. Also, this will never happen beyond the database facade `DBAccess`. The following code deploys `log` over execution of *`block`*:

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

The scoping strategy specifies that (a) the aspect sees all join points in the dynamic extent of the block except when the target of a call is of type `DBAccess`; (b) the aspect is captured in all created `Reservation` 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 `Date`. Scoping strategies subsume existing proposals and enhance aspect reusability by giving fine-grained control over the scope of dynamically-deployed aspects (previous work [25] details why scoping strategies cannot be expressed with a combination of other existing aspect scoping mechanisms). Furthermore, scoping strategies are applicable to variable bindings, as well as other kinds of program adaptations than aspects [26]. The expressiveness of scoping strategies (in terms of macro-expressibility [10]) for variable bindings is discussed elsewhere [26].

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, scoping strategies cover that space, but are formulated in a non-distributed context. This work therefore explores support for expressive scoping of distributed aspects, by proposing means to augment the power of scoping strategies to express distribution-related constraints on the scope of deployed aspects.

---

[2] 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 [23].

```
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 ...
  }
}
```

Fig. 1. Section of the travel service: booking of a package.

2.4. Expressive scoping scenarios

We now present several distributed scenarios for which existing languages provide insufficient deployment support. We informally describe a number of scoping strategies that solve these scenarios. Section 3 discusses our attempts to implement these scenarios using AspectJ, CaesarJ and AWED, showing where all of these languages fall short.

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 and sends these to the client using a callback. This allows a combination of reservations to be selected: the travel agent picks items from a list, and these are returned to the server. An outline of the travel server code for this scenario is shown in Fig. 1. The `trv` and `sel` parameters of the `bookPackage` service contain relevant information on the traveler and the object used as a callback to select reservations, respectively. Fig. 2 illustrates the reservation confirmation phase, with deployment of a billing aspect (discussed next) and a privacy aspect (discussed in 2.4.3).

2.4.1. Case 1: Controlling propagation

In this system, we want to implement a `Billing` aspect, using an implementation of the wormhole pattern [13], 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). To update their internal bookkeeping they then call a `resConfirmed` method on themselves, the argument type of which is a superclass of `FlightsInfo` and `RoomsInfo`. The name and parameter list of this method is fixed by convention, which allows the `Billing` aspect to apply.

The above example has some issues that should be addressed: (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 specify the deployment to use a specific scoping strategy that cuts propagation at these points.

2.4.2. Case 2: Controlling activation

Suppose that performance information needs to be gathered from the client. We wish to reuse an existing `Profiling` aspect (with a generic `execution(* *.*(..))` pointcut). This requires the aspect to be deployed with a scoping strategy

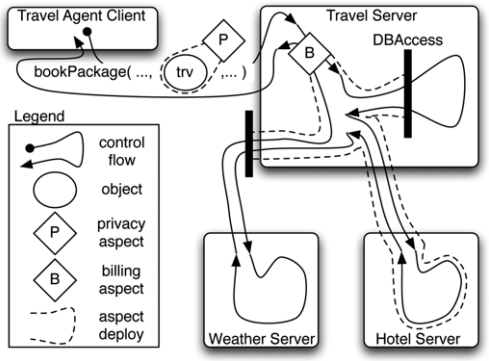

Fig. 2. Travel agent system with privacy and billing aspects deployed. The illustration focuses on the `confirm` phase, *i.e.,* after the callback to the travel client returned a specific reservation to confirm.

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.

Note that this 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 shows a list that allows the travel agent to pick among proposed reservations for a given package. We also want to gather profiling information for this code.

2.4.3. 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. While this data cannot be modified due to lack of setter methods, for privacy reasons the client may not reveal any sensitive information to untrusted hosts. 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 (as indicated by `...` in the pointcut) to return `"N/A"`. It should only be active when the object does not reside on a trusted host. Ideally this is defined by considering groups of hosts, e.g. a `TravelGroup` that contains all hosts that are trusted. Therefore, the aspect should be active when outside of the `TravelGroup`.

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();
```

Here, similarly to statically-scoped aspects in AspectScheme, we can deploy the aspect in the factory. The deployment of a statically-scoped aspect implies that all objects created by the deployment target also have the aspect deployed. As a consequence, all objects created by the factory also have the privacy aspect deployed. We however want a more specific kind of deployment: the aspect should only propagate to `Traveler` objects created by the factory, not to all objects that the factory creates. By being embedded in such `Traveler` objects, the aspect follows them as they are sent over the network, as depicted on Fig. 2. In addition, the scoping strategy should specify that the aspect is only active outside the `TravelGroup`.

3. Tentative implementation

Having introduced the three scenarios that we wish to support, we now detail how we can attempt to implement these in AspectJ (Section 3.1), CaesarJ (Section 3.2), and AWED (Section 3.3). Recall that the focus of this work is to provide expressive scoping through the use of scoping strategies. Neither of these three languages have support for scoping strategies. We therefore need to achieve the required behavior by including scoping in either the pointcut or the advice of the aspect. Including scoping conditions in either of these will however reduce their reuse possibilities. We attempt to achieve the highest reuse potential by limiting changes to the pointcut definitions, as well as by structuring these pointcuts with reuse in mind. Nonetheless, 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 [25]. In contrast, our proposal, introduced in Section 6, does concisely express the strategies we desire.

3.1. AspectJ implementation

The first hurdle we encounter in implementing the three scenarios is that AspectJ does not provide any support for dynamic deployment, as we discussed in Section 2.1. This forces us to make the first major assumption in this implementation:

> **A1:** We need to be able to weave all affected classes and update them in a consistent way across the entire distributed system. As this is a difficult problem,[3] we fear that in practice this boils down to being able to stop, weave, and restart the entire system.

3.1.1. Case 1: Controlling propagation

AspectJ does not provide for any control on propagation of aspects, as their scope is the entire woven system. As a result we need to simulate propagation control by selectively activating the aspect, adding some extra condition to its pointcut. This is performed by a `toBill(inf)` test, whose implementation we will discuss shortly, determining whether the aspect is in scope or not.

```
aspect Billing {

  after(TravelInfo inf) returning: billMe(inf) {
    bill.addItem(inf);
  }

  pointcut billMe(TravelInfo inf):
    execution(* *.resConfirmed(TravelInfo)) && args(inf) && if(toBill(inf));

  [... toBill implementation ...]
}
```

The `inScope()` pointcut. There is an important downside to the above pointcut when considering reuse, which is that this pointcut is tangled. It addresses two concerns: whether the aspect is in scope, and if so, whether it applies. Separating these two concerns in two pointcuts, e.g. into `billMe` and `inScope`, would ease reuse of the aspect. In this setup only the `inScope` pointcut needs to be changed if the scope is different. The following code achieves such a modularization:

```
aspect Billing {
  after(TravelInfo inf) returning: billMe(inf) && inScope () {
   bill.addItem(inf);
  }

  pointcut billMe(TravelInfo inf):
    execution(* *.resConfirmed(TravelInfo)) && args(inf);

  private pointcut inScope2(TravelInfo inf):
   execution(* *.resConfirmed(TravelInfo)) && args(inf) && if(toBill(inf));

  private pointcut inScope(): inScope2(TravelInfo);

  [... toBill implementation ...]
}
```

Achieving the separation of the original pointcut and the scoping specification leads to code which is somewhat convoluted, requiring an extra `inScope2` pointcut. The goal of this last pointcut is to perform the `toBill` test. For this, it needs to repeat the `execution` pointcut of `billMe`, purely to get the argument using `args(inf)`. `inScope2` however also needs to declare the type of `inf` to be `TravelInfo`, yielding an `inScope2(TravelInfo inf)` signature. The goal of `inScope` is then to simply hide the `TravelInfo` parameter from the `inScope2` pointcut, abstracting away from these implementation details.

It is clear that the advantage in modularization comes at a cost of more code. We expect however that this extra code will not cause much overhead at runtime as this is an easy case for optimization by the AspectJ compiler. Therefore, in the remainder of this section we use such modularized pointcuts.

A distributed control flow pointcut. The `toBill` method should test whether the call to `resConfirmed` happens within the distributed control flow of the `confirm(res)` call, excluding hosts outside the travel group, as well as the control flow beyond the database facade. However, as the AspectJ `cflow` pointcut does not provide any support for distribution, there is no straightforward way to implement this.

[3] See [30] for a detailed introduction to atomic weaving of aspects in distributed systems.

The implementation of a general distributed cflow construct for AspectJ is arguably non-trivial. We choose not to take this route but instead implement the simplest solution for this specific case. Our solution relies on a property of the `res` parameter of the `confirm` call: it contains all the `TravelInfo` objects that will be the arguments of the `resConfirmed` calls in the control flow. We tag all these objects with an extra flag, and `toBill` simply tests whether this flag is set or not. If it is, we are in the distributed control flow, if not, we are not.

```
aspect Billing {
  [... code detailed above ...]

  private static boolean toBill(TravelInfo inf) {
    return inf.toBill;
  }

  private boolean TravelInfo.toBill = false;




  Booking around(Reservation res) :
    execution(Booking TravelService.confirm(Reservation)) && args(res) {
    [... for all TravelInfo objects in res toBill = true ...]
    Booking b = proceed(res);
    [... for all TravelInfo objects in res toBill = false ...]
    return b;
  }
}
```

This yields the second major assumption for this implementation:

A2: For all method executions in the distributed control flow that we wish to capture, there is at least one parameter `obj` that exists at the beginning of the control flow and is reachable at that point.

This assumption implicitly relies on the following:

A3: (A copy of) the above parameter `obj` is not stored within this control flow and later retrieved, nor is it concurrently accessed.

Both assumptions hold in this application, however it is clear that we cannot expect this to be so in the general case.

Restricting the distributed control flow. To conclude the implementation of this case, we need to simulate stopping propagation when leaving the travel group or when crossing the database facade. Our distributed control flow pointcut however does neither. To be inactive outside the travel group we modify the `inScope` pointcut. We include a test for whether the current host is part of the travel group, assuming the aspect implements an `inTravelGroup()` method for this:

```
private pointcut inScope(): inScope2(TravelInfo) && if(inTravelGroup());
```

Deactivation beyond the database facade is obtained by toggling the `toBill` flag at the `DBAccess` facade, as follows:

```
aspect Billing {
  [... code detailed above ...]

 void around(TravelInfo inf) :
   call(* DBAccess.*(..)) && args(inf) && if(inf.toBill) {
   boolean wasSet = inf.toBill;
   if (wasSet) inf.toBill = false;
   proceed(inf);
   if (wasSet) inf.toBill = true;
  }

  [... inTravelGroup implementation ...]
}
```

3.1.2. Case 2: Controlling activation

Intuitively, it seems that we can easily restrict profiling to the client by only weaving the classes that are present on the client. This is indeed possible, but relies on a subtlety of how Java RMI works in combination with AspectJ. The origin of this complication is that we pass objects between the client and the server, e.g. in the `bookPackage` call. In this call, the first three arguments (a `FlightSpec`, a `RoomSpec` and a `Traveler`) are passed by copy. A Java RMI call with pass-by-copy arguments or return value requires that the classes of these values are identical on both hosts. This is determined by the contents of

a `classVersionUID` field [23]. If we do not declare this field, it is automatically generated at compile time, and thus will be different on the client and the server, due to the AspectJ weaving process. As a result, the RMI call will throw a class version exception. If we do declare and initialize this field, it will be the same on both client and server, and RMI will consider both the woven class (on the client) and unwoven class (on the server) as being the same.

While this solution works, it is low-level and brittle, because it relies on unspecified guarantees of the current implementation of AspectJ with respect to weaving and serialization. Therefore, we prefer a different solution to restricting profiling to the client that is more robust. Similar to the solution in the first case, we obtain activation of the profiling aspect to the travel client by adding an `inScope()` condition to its pointcut:

```
aspect Profiling {
  pointcut Profile(): execution(* *.*(..));

  private pointcut inScope() :
    if(inTravelClient())
    && !cflow(execution(* Profiling+.inTravelClient()));

  boolean inTravelClient(){ [...] }

  before(): profile() && inScope(){ [...] }

  after(): profile() && inScope() { [...] }
}
```

The `inTravelClient()` method of the aspect determines if the current host is the travel client. Due to the fact that the Profiling aspect uses a generic `execution(* *.*(..))` pointcut, we need to ensure that the `inTravelClient()` method itself is not profiled, otherwise an infinite loop occurs. Hence the last condition of the pointcut.

3.1.3. Case 3: Controlling per-object activation

Similar to the lack of distributed control flow, AspectJ does not provide support for deploying an aspect on specific objects in a distributed setting. An aspect deployed on an object (using `perthis` or `pertarget`) on one host does not migrate with the object when the latter moves from one host to another host. Any state that is held in the aspect is lost when an object migrates. As a result, outside of the travel service client we cannot distinguish between an object that had the `Privacy` aspect deployed at instantiation time and one that did not.

Such an object-specific deployment could be simulated as follows: in the `Privacy` aspect we use an inter-type declaration to add two extra flags to the objects, similar to what we did for the `Billing` aspect. One flag specifies whether the object should have the `Privacy` aspect embedded, and the second whether the aspect should be active or not. We then have the `Privacy` aspect activate whenever both flags are set. We can use advice to set and clear flags when required, again as in the `Billing` aspect. The problem with this is however that the pointcut for this flag manipulation advice can arguably become non-trivial.

To avoid an overly complex pointcut for flag manipulation we do not implement the general case. Instead we restrict ourselves to an implementation specific to this example: all `Traveler` objects created on the client are protected if they are out of the travel group. We add the current host name to `Traveler` objects at creation time. The `inScope` pointcut then verifies if this is the client host name and whether it is running outside the travel group, using an `outOfTravelGroup` method.

```
public aspect Privacy {
  private String Traveler.creationHost = getHostName();

  private pointcut clientCreated(Traveler t):
    this(t) && if(t.creationHost.equals("TravelClient"));

  private pointcut inScope():
    clientCreated(Traveler) && if(outOfTravelGroup());

  private pointcut protectMe() : [...]

  String around() : protectMe() && inScope() {
    return "N/A";
  }

  [... implementation of getHostName and outOfTravelGroup ...]
}
```

3.1.4. Summary of limitations

Using AspectJ we are able to obtain a correct implementation of all three cases, but there are however significant downsides. Not only do we need to modify the pointcuts of all aspects, but we also need to take into account the three far-reaching assumptions we highlighted above: allow for static deployment (**A1**), have specific objects follow the entire distributed control flow (**A2**), and only this control flow (**A3**). It is clear that in the general case we cannot make such assumptions. Therefore AspectJ fails to provide adequate support for these scenarios.

3.2. CaesarJ implementation

We now consider the CaesarJ language [1]. Unlike AspectJ, CaesarJ provides support for both *dynamic* and *remote deployment*.

Dynamic deployment (recall Section 2.2) refers to the possibility of dynamically deploying and undeploying aspects either locally inside the current JVM or within a thread. In CaesarJ, aspects are explicitly instantiated from classes.[4] An aspect instance *asp* can be deployed locally using the construct `deploy` *asp*. It can then be undeployed using the dual construct `undeploy` *asp*. The scope of the aspect instance can also be further restricted either by deploying the aspect on specific objects with the API calls `deployOnObject(`*asp*`,` *obj*`)` and `undeployOnObject(`*asp*`,` *obj*`)`, or by using the construct `deploy(`*asp*`)`*block*, which deploys the aspect *asp* along the control flow delimited by the block *block*.

Remote deployment enables interception of *remote join points*. Remote deployment does not occur by default but requires that, first, remote deployment is activated on each remote host and, second, each aspect instance *asp* is deployed on the appropriate remote host *host* through the construct *host*`.deployAspect(`*asp*`)`.

In the case of thread-based deployment, when the control flow jumps to a remote host, the aspect is automatically deployed on the remote host without the need for explicit remote activation. This deployment along a *distributed control flow* combines dynamic and remote deployment.[5]

As a result, CaesarJ may look like an ideal target for implementing the scenarios associated to our running example. As we shall see, the facilities provided by CaesarJ are an interesting first step but are far from providing solutions to all the issues raised by these scenarios.

3.2.1. Case 1: Controlling propagation

Basic scenario. As long as we do not restrict propagation of the billing aspect along the control flow of the call to the `confirm` method, the deployment of the `Billing` aspect can be straightforwardly implemented using distributed control flow:

```
Booking booking = null;
deploy (new Billing()){ booking = confirm(res); }
```

The `Billing` aspect has just to be adapted to cater for the syntax of CaesarJ. Concretely, the keyword `aspect` has simply to be replaced by the keyword `cclass` to tell the compiler that it is not a plain Java class.

Refined scenario. Refining the scenario in order to avoid aspect propagation within the weather server and behind the database facade is however an issue. There is no way to explicitly refine deployment by excluding part of the control flow, based either on host information (to restrict propagation to the travel group) or on a receiver's type (to deactivate propagation beyond the database facade). Again, like with AspectJ (Section 3.1), the aspect itself, pointcut or advice, has to be modified.

Avoiding the selection of join points within the database can be obtained by defining an `inScope` pointcut, as we have done before. This pointcut excludes the control flow beyond the database facade:

```
private pointcut DBBarrier(): call(* DBAccess.*(..));
private pointcut inScope() : !cflow(DBBarrier());
```

Propagation within the weather server could be excluded in the same way, but this does not take into account the general intention of the scenario, which is to exclude propagation within members of the travel group. Unfortunately, it is not possible to use conditional pointcuts, as CaesarJ does not provide support for them, or even test host information as part of the advice code. The latter is because the piece of advice part of the billing aspect is always executed on the travel server. When the aspect propagates outside of the travel server, the host information required to know whether the execution is outside of the travel group is where the join points take place. This is how a remote aspect works: the piece of advice is executed locally (here, on the travel server) and the join points remotely (along the control flow of package booking, see Fig. 2) . Whereas distinguishing the locus of execution of a join point and a piece of advice does not make sense in a non-distributed context, it is typical of distributed applications. The problem is that (i) host information cannot be tested

[4] This classes are distinguished from plain Java classes through the keyword `cclass` and contain at least a pointcut and its associated piece of advice.

[5] This corresponds to the way this is implemented in the current version of CaesarJ (version 0.9.0), using the same syntax as thread-based deployment. Previously, a specific API call had to be used [1].

at the pointcut level as there is no specific pointcut to do so and conditional pointcuts are not available in CaesarJ (ii) host information is not part of the reflective information about the current join point made available to advice code.[6]

As a result, the general scenario excluding propagation outside the travel group cannot be implemented without changing the base program (by, for instance, enforcing the addition of a parameter carrying host information to remote methods). A fallback consists of making the following assumption:

> **B1:** The weather server is the only server outside of the travel group.

This makes it possible to handle the weather server like the database, with a specific `inScope` condition:

```
private pointcut WeatherBarrier(): execution(* IWeatherService.*(..));
private pointcut inScope() : !cflow(WeatherBarrier);
```

3.2.2. Case 2: Controlling activation

As discussed above, there is no way to deactivate aspects along a distributed control flow without significantly modifying the base program. This calls for a new assumption:

> **B2:** The travel agent client is not multi-threaded and does not perform other computations than the ones we are interested in.

Then, it suffices to deploy the `Profiling` aspect on the local JVM. If this was not the case, the pointcut would have to be extended with a new `inScope` condition restricting profiling to the local control flows of interest:

```
pointcut cflowRoots():
  execution(static void Main.main(..))         // client entry point
  || execution(Reservation Selector.pick(..)); // callback entry point
pointcut inScope(): cflow(cflowRoots());
```

3.2.3. Case 3: Controlling per-object activation

It may look like CaesarJ offers all the building blocks to implement this scenario in a general way: per-object deployment and remote deployment. Unfortunately, CaesarJ is limited in that these two features cannot be composed: per-object deployment does not apply remotely. It is not possible either to use the fact that the `Traveler` objects of interest are only used along a specific distributed control flow as we cannot distinguish, along this control flow, the trusted and the untrusted servers. Finally, in the absence of inter-type declarations and conditional pointcuts, it is not possible to implement the specific AspectJ solution that was described previously.

3.2.4. Summary of limitations

CaesarJ offers a number of interesting features: per-object deployment, deployment on a JVM or on a (distributed) control flow, and remote deployment. Unfortunately, these features cannot be composed. Per-object deployment does not apply remotely and, although an aspect can be deployed remotely on a host by host basis, the activation of an aspect along a distributed control flow is independent from the host. The lack of conditional pointcuts and inter-type declarations further complicates the implementation of workarounds. As a whole, there is no way to directly capture the intentions of our various scenarios. Implementing case 1 requires hardwiring the definition of the travel group (**B1**) and using scoping pointcuts. Implementing case 2 requires either making assumptions on the implementation of the travel agent client (**B2**) or using scoping pointcuts based on a precise knowledge of the application control flow. Even worse, the impossibility of combining per-object and remote deployment linked to other incidental shortcomings prevents a reasonably simple implementation of case 3.

3.3. AWED implementation

AWED does not include means for the dynamic deployment of aspects: aspects are deployed statically on sets of hosts. In contrast to AspectJ and CaesarJ, AWED provides three features that allow pointcuts and advice to be defined declaratively and concisely over sets of hosts: remote cflow and sequence pointcuts, relative host selectors and host groups. *Remote cflow and sequence pointcuts* enable the matching of control-flow and sequence relationships between execution events occurring on different hosts. *Relative host specifications* allow hosts (on which execution events are to be matched or advice to be executed) to be defined relative to where an aspect is deployed. This mechanism makes it possible, *e.g.* to differentiate between local and remote hosts. *Host groups*, finally, provide a simple mechanism to quantify over many hosts in pointcut and advice definitions.

Overall, these means for expressive pointcut and advice declarations complement AWED's static strategy for the deployment of aspects on sets of hosts such that distributed aspects directly express relationships and modifications

[6] It is actually worse than that as, currently, this reflective information is not serializable, which means that the information exposed by the remote join points (for instance their signature) is not available. This is quite illustrative of the kind of "details" that have to be taken care of when switching from a sequential to a distributed setting.

involving all relevant machines. Nonetheless we still need to make the same assumption (**A1**) as in AspectJ, which is that we allow for static deployment.

3.3.1. Case 1: Controlling propagation

Applying the billing aspect as described in the first scenario can be expressed concisely using the remote control-flow and host pointcuts of AWED. Compared to the AspectJ implementation, AWED cflow pointcuts are not subject to limitations **A2** and **A3** from Section 3.1. As a consequence, this scenario can be straightforwardly implemented as shown below:

```
aspect Billing {
  pointcut billMe(TravelInfo inf):
    execution(* *.resConfirmed(TravelInfo)) & args(inf);
  pointcut inScope():
    cflow(TravelService.confirm(Reservation))
    && !cflow(DBAccess.current())
    && host(TravelServer, HotelServer);

  after(TravelInfo inf): billMe(inf) && inScope() {
    bill.addItem(inf);
  }
}
```

Here, the calls beyond the database facade are excluded by the second term within the `inScope()` conjunction. The calls on the weather server are not taken into account because the weather server's host is not included in the host list given on the third line. Note that here a negated cflow condition on the call to the weather server could have been used as well. This is because the scope of calls originating in the weather server do not extend to other hosts.

3.3.2. Case 2: Controlling activation

Activation of aspects, e.g., in order to restrict the application of a profiling aspect to client computations only can be straightforwardly expressed in AWED by combinations of cflow pointcuts and host-selecting pointcuts. The robust variant for the profiling activation presented in Section 3.1 is applicable directly to the distributed setting in AWED using the following scope definition:

```
pointcut inScope():
  cflow(execution(static void Main.main()))  // client entry point
    && host(TravelClient)
```

This pointcut definition can be used in aspects deployed on any host and ensures that only calls on the client host `clientH` are profiled. Note that this formulation is more concise than the AspectJ and CaesarJ solutions discussed above.

3.3.3. Case 3: Controlling per-object activation

Hiding personal data from computations on certain hosts can also be expressed in a direct manner using AWED through the use of a sequence pointcut as below:

```
pointcut protectMe(Traveler trv): seq(
   call(Traveler.Traveler(..)) && this(TravelerFactory)
      && target(trv) && host(TravelClient),
   (execution(String Traveler.getAdress())
        || execution(String Traveler.getPhone()))
      && target(trv) && !host(TravelGroup));
```

This definition replaces the `protectMe` pointcut of the original aspect. It specifies a sequence of operations that matches calls to the methods `getAddress` and `getPhone` that occur after a traveler object has been created by the factory on the client and the same object is used outside of (*i.e.,* not on a host in) the group `TravelGroup`. This definition closely matches the AspectJ solution described in Section 3.1 in that it makes explicit the creation of the factory on the client host and only restricts accesses outside the travel group. However, the AWED solution does not need helper methods encoding conditions on the distributed state of the overall system.

While this sequence pointcut concisely expresses the activation condition for implementing the privacy aspect in our running example, it does not match the underlying objectives in two ways. First, since aspects cannot be bound to individual objects (and not be deployed on a per instance manner), the AWED aspect implements external control exerted by the execution infrastructure instead of a control mechanism implemented as part of the object itself. Second, the above aspect can be used as an additional scoping condition for the previously considered control-flow pointcuts, but provides a complete definition of the necessary relationships between execution events.

Table 1
Non-distributed base language and aspects.

Base program syntax
`Definition ::= (define Variable Expression)`
`Expression ::=`
`Constant`
`| Variable`
`| (lambda (Variable*) Expression)`
`| (Expression Expression*)`
`| (set! Variable Expression)`

Aspect program syntax
`Aspect ::= (Pointcut . Advice)`
`Pointcut ::= Expression`
`Advice ::= Expression`

Join point structure `(define-struct jp (fun args parent))`
Pointcut domain $\mathcal{PC} = \mathit{JoinPoint} \rightarrow \mathit{Bool}$
Before advice domain $\mathcal{ADV} = \mathit{JoinPoint} \rightarrow \mathit{Unit}$
Aspect domain $\mathcal{ASP} = \mathcal{PC} \times \mathcal{ADV}$
Scoping strategy syntax `Strategy ::= <Expression,Expression,Expression>`
Scoping strategy domain $\mathcal{S} = \mathcal{PC} \times \mathcal{PC} \times \mathcal{PC}$

Aspect deployment primitives
`Expression ::= ... | Deploy`
`Deploy ::=`
`(deploy Strategy Aspect Expression)`
`| (deploy-on Strategy Aspect Expression)`

Aspect Environment domain $A = \{(a, \sigma) \mid a \in \mathcal{ASP},\ \sigma \in \mathcal{S}\}$

3.3.4. Summary of limitations

AWED performs significantly better than AspectJ and CaesarJ, only requiring the assumption (**A1**), which is that we allow for static deployment. This is thanks to support for distributed control flow, host matching and sequence pointcuts. Nonetheless, we are still confronted with reuse issues: the lack of dynamic deployment requires rewriting the pointcuts of the respective aspects, and the use of sequence pointcuts hardcodes sequences of events in the pointcuts themselves. The use of scoping strategies overcomes these limitations, as we detail in the remainder of this paper.

4. Scoping of non-distributed aspects

In this section, we introduce scoping of non-distributed aspects. First, we present our base language that is based on Scheme. Then, we define aspects in terms of Scheme expressions and aspect weaving using a corresponding interpreter. Finally, we introduce scoping strategies as a means to restrict the scope of aspects.

4.1. Base language

Our base language, shown in Table 1, is a subset of Scheme. Expressions are constants (*e.g.*, numbers, strings, booleans), variables and function definitions, and applications that use call-by-value argument passing. Variables are mutable. The only data structure supported are cons cells, and by extension, n-ary tuples and lists. In addition, we assume the language supports a set of standard primitives inherited from Scheme itself.

The operational semantics of our base language is defined by the interpreter[7] in Fig. 3. This interpreter is written in environment-passing style [11]: the main function, `eval`, has two arguments: an expression (*i.e.,* an abstract syntax tree) and an environment `E` that binds variables to values. The interpreter evaluates an expression following a simple case-based test. When the expression to be evaluated is a constant (tested by the predicate `const?`), then its value is returned by the accessor `const-value`. 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. Finally, applying a function evaluates the body of the closure in its definition-time environment that is extended with new bindings for the formal parameters.

4.2. Aspects

We now extend our higher-order base language with 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 [9]). The focus of our work is on

[7] Our executable Scheme interpreters, along with examples, are available online: http://pleiad.dcc.uchile.cl/research/scope.

```
;; evaluate expression exp in lexical environment E
(define (eval exp E)
  (cond ((const? exp) (const-value exp))
        ((var? exp)   (lookup (var-name exp) E))
        ((fun? exp)   (make-closure (fun-params exp) (fun-body exp) E))
        ((app? exp)   (let* ((cl   (eval (app-fun exp) E))
                             (args (eval-args (app-args exp) E))
                             (env  (extend-env (closure-params cl) args
                                               (closure-env cl))))
                        (eval (closure-body cl) env)))
        ...))
```

Fig. 3. Interpretation of a higher-order procedural language.

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

(define || (lambda (pc1 pc2) (lambda (jp) (or (pc1 jp) (pc2 jp)))))

(define within (lambda (f) (lambda (jp)
  (and (has-parent? jp) ((call f) (jp-parent jp))))))

(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))))))
```

Fig. 4. Some typical pointcut designators.

scoping, that is, how to delimit the set of join points that an aspect can potentially match; the kinds of join points and effects at these join points is an orthogonal concern. So, similar to our previous work [25], for the sake of simplicity and without loss of generality, we restrict ourselves to function call join points, functional pointcuts and before advices in the following.

Join points correspond to function applications in the base program. A function call is either top-level or nested within pending function calls. So, 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). In our interpreter 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))
```

A pointcut is a predicate over join points in context, *i.e.,* it is a function of type[8]:

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

We can then define combinators that compose pointcuts. A pointcut designator, such as `call`, is a function that returns a pointcut. Fig. 4 shows definitions of typical pointcut designators. The function `call` takes a function identifier `f` as a parameter and returns as pointcut a function that takes a join point `jp` and tests if the join point corresponds to `f`. For instance, if `reserve` is a function in scope, then `(call reserve)` returns a pointcut that matches applications of that function. The function `||` returns a pointcut that checks if a join point satisfies a disjunction of pointcuts. Pointcut designators can also inspect the call stack of join points. The function `within` checks the last pending function call. It takes a function identifier `f` as a parameter and returns a function that takes a join point `jp` and tests if the join point nesting `jp` is a call to `f`. The function `cflow` tests all pending function calls. It takes a pointcut `pc` as a parameter and returns a new pointcut that tests if the given join point satisfies the pointcut `pc`, or if one of the nesting join points satisfies that pointcut (by calling `cflowbelow`). Finally, the function `cflowbelow` is similar to `cflow` but is restricted to the parents of the join point.

We simply model advice as functions of type[9]:

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

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.

[8] This formalization is simplified, because we do not account for context exposure of pointcuts. Context exposure can, however, be integrated using well-known means, in particular by changing the signature of a pointcut to return either false (no match), or an environment containing bindings to be provided to the advice [17,9].

[9] This definition reflects the simplifications we have mentioned previously: pointcuts do not expose context information, and we only support before advice. Accounting for both context information and around advice with `proceed` would have to follow the formalization of [9].

```
;; the global aspect environment
(define *aspects* '()) ;; populated upon program elaboration

;; evaluate expression exp in lexical environment E
;; with current join point jp
(define (eval exp E jp)
  (cond ((const? exp) (const-value exp))
        ((var? exp)   (lookup (var-name exp) E))
        ((fun? exp)   (make-closure (fun-params exp) (fun-body exp) E))
        ((app? exp)   (let* ((cl   (eval (app-fun exp) E jp))
                             (args (eval-args (app-args exp) E jp))
                             (njp  (make-jp cl args jp))
                             (env  (extend-env (closure-params cl) args
                                               (closure-env cl))))
                        (weave-all njp)
                        (eval (closure-body cl) env njp)))
        ...))

;; weaves all aspects
(define (weave-all jp)
  (map (lambda (asp) (weave asp jp)) *aspects*))

;; weaves aspect on jp (if pc match, apply advice)
(define (weave asp jp)
  (if (app/prim (asp-pc asp) jp)
      (app/prim (asp-adv asp) jp)))

;; app/prim: primitive application, does not generate join points
```

Fig. 5. Simplified pointcut-advice interpreter.

Finally, an aspect $a \in \mathcal{ASP}$ is simply represented as a pair of a pointcut and an advice. To construct an aspect, a dotted pair notation is supported in the language (see Table 1, aspect program syntax).

Aspect weaving is defined by the interpreter outlined in Fig. 5. It extends the interpreter of the base language with a third argument `jp`: the join point at the enclosing function application. This argument enables the representation of join points in context. When a function is to be applied, the interpreter evaluates the function and its arguments (as the base interpreter) but also creates a new join point `njp` representing that application. Then it triggers weaving, and it proceeds with executing the function application, with the new join point. In our model both pointcuts and advices are *first-class values*. However, we consider that pointcuts and advices are executed by `app/prim` in "sandboxes" (*e.g.*, by the base language interpreter) where no aspect can match, thereby avoiding 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 the work of Tanter [24,27].

At this stage, we simply consider a global aspect environment, *i.e.,* a global variable in the interpreter. Weaving 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.

4.3. Non-distributed scoping strategies

Aspect scoping strategies have initially been proposed in a non-distributed context [25] under the name of deployment strategies. Compared to the environment-passing style interpreters we have presented until now (Fig. 5), an interpreter of aspects with dynamic deployment and scoping strategies evaluates an expression within an *aspect environment* that is passed around between evaluation steps.

An interpreter for non-distributed scoping strategies is shown in Fig. 6. It takes an extra parameter, the aspect environment `A` that contains the currently-deployed aspects whose pointcuts must be evaluated. It is a set of pairs $\langle a, \sigma \rangle$, where a is an aspect and σ is a scoping strategy. In brief, a scoping strategy $\sigma \in \mathcal{S}$ is a triple of pointcuts $\langle c, d, f \rangle$, where c and d are *propagation functions* used to specify, respectively, call stack propagation and propagation within entities that are subject to delayed evaluation (*e.g.*, created functions or objects), and f is a *join point filter* used to express deployment-specific filtering of the join points that are visible to the deployed aspect. All three components are pointcuts, *i.e.,* they take a join point as parameter and return a boolean (see Table 1).

An aspect is initially inserted into the environment when it is deployed, along with its strategy defined as a 3-tuple of pointcuts `<Call,Delay,Filter>`. An expression (`deploy` $\langle c, d, f \rangle\ a\ e$) is evaluated by the third case in the interpreter, `(depl? exp)`. It inserts $\langle a, \sigma \rangle$ in the current aspect environment before proceeding with the evaluation of the reducible expression e.

```
(define (eval exp E jp A)
 (cond ((const? exp) (const-value exp))
       ((var? exp)   (lookup (var-name exp) E))
       ((depl? exp)  (let ((dasp (make-dasp exp E A jp)))
                       (eval (depl-body exp) E jp (cons dasp A))))
       ((fun? exp)   (make-closure (fun-params exp) (fun-body exp) E
                                   (collect-match-d jp A)))
       ((app? exp)   (let* ((cl   (eval (app-fun exp) E jp A))
                            (args (eval-args (app-args exp) E jp A))
                            (njp  (make-jp cl args jp))
                            (env  (extend-env (closure-params cl) args
                                              (closure-env cl)))
                            (asps (union (collect-match-c njp A)
                                         (closure-aspects cl))))
                      (weave-some A njp)
                      (eval (closure-body cl) env njp asps)))
       ...))

(define (weave-some A jp) (weave-all (collect-match-f A jp) jp))
(define (collect-match-c jp asps)
  (collect-if (lambda (a) ((dasp-c a) jp)) asps))
(define (collect-match-d jp asps)
  (collect-if (lambda (a) ((dasp-d a) jp)) asps))
(define (collect-match-f jp asps)
  (collect-if (lambda (a) ((dasp-f a) jp)) asps))
```

Fig. 6. Interpretation of scoping strategies for a higher-order functional language.

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

Fig. 7. Semantics of non-distributed scoping strategies in a nutshell.

When a function is evaluated by the fourth case of the interpreter, `(fun? exp)`, a closure is built by `make-closure`. This closure stores aspects that are selected from the aspect environment by the function `collect-match-d`, which uses the delay component d of the scoping strategy. When a function application is evaluated by the fifth case of the interpreter, `(app? exp)`, a new aspect environment `asps` is computed first. It contains the aspects selected from the aspect environment by the function `collect-match-c`, which uses the call stack component c of the scoping strategy. It also contains the aspects stored in the closure `cl` of the function to be applied. Next, the function `weave-some` first selects the aspects that should potentially be woven by using the filter f of their scoping strategy (`collect-match-f`). The interpreter weaves the resulting aspects that match the current join point. Finally, the body of the applied function is evaluated in the context of `asps`.

Fig. 7 summarizes the semantics of scoping strategies in a non-distributed context:

- 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.

The above description of scoping strategies is directly based on Tanter's original formulation [25]. For this work on distributed aspects, we have found practical and necessary to extend the set of deployment expressions to include the ability to deploy an aspect on an *existing* procedural value using (`deploy-on` $a\ \sigma\ e$). Intuitively, e is first reduced to a procedural value, and aspect a is deployed *within that value*, with scoping strategy σ. This means that a is deployed over the evaluation of the function body each time the function is applied, with scoping strategy σ.

Supporting explicit per-object deployment in the language is direct. It is sufficient to add the following case in the interpreter of Fig. 6:

```
((depl-on? exp)
  (let* ((cl   (eval (depl-on-fun exp) E jp A))
```

Table 2
Distributed base language and aspects.

Distributed base program syntax

```
Expression ::=
    ...
    | (export Expression)
    | (export-as Literal Expression)
    | (lookup Host Literal)
```

Distributed join point structure

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

Aspect deployment primitives

```
Deploy ::=
      ...
    | deploy-global-on-host Host Aspect
```

```
      (dasp (make-dasp exp E A jp)))
  (set-closure-aspects! cl (union (list dasp)
                                  (closure-aspects cl)))))
```

To interpret the `deploy-on` expression, the interpreter first obtains both the closure and the aspect (with its associated scoping strategy), and then adds the aspect in the aspect environment of the closure (using `union` to avoid duplicates). Although not included in the core of the article, `deploy-on` was discussed in [25] (in an object-oriented setting); in particular, it is argued that `deploy-on` is more powerful than per-instance aspect deployment of 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 a fixed scoping strategy $\sigma = \langle \lambda jp.\texttt{false},\ \lambda jp.\texttt{false},\ \lambda jp.\texttt{true} \rangle$).

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 adding distribution to the small Scheme-like higher-order procedural language introduced in Section 4, then aspects, and finally add some typical distributed aspect support. This model is further extended in Section 6 to fully support distributed scoping strategies.

5.1. Adding distribution

We now extend the language with support for distribution as shown in Table 2. 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.

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

Fig. 8 sketches the evolution of the interpreter to support distribution. The interpretation of a function application by `eval-app` now needs to discriminate between local and remote calls. Local calls are interpreted by `eval-exec` as before. Remote calls interpreted by `remote-exec` imply extracting information from the stub, serializing it and sending it to the target host. 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, the function `receive-call` looks up the actual closure, and evaluates it locally. The result is then serialized and sent back to the caller.

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

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

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

Fig. 8. Interpretation of a higher-order procedural language with distribution.

5.2. Distributed aspects

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 and AWED's remote advice execution, 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 aspects 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 scoping strategies.

When aspects are passed over the network, for instance when they are deployed on a remote host, like above, they are treated as plain values, passed by copy. But since pointcuts and advice are first-class functions, they can also be *remote* functions: 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.

```
(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))))))
```

Fig. 9. Distribution-related pointcut designators.

Because a join point in context keeps a parent link to its predecessor join point, it is an abstraction of the call stack (Section 4.2). 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. 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.

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 [3,19,29], *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. 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 [29]. This allows the possibility to define host groups like in AWED [3].

Fig. 9 presents a number of pointcuts and pointcut designators that take advantage of these extensions. The pointcut designator `host` matches a join point only if its host matches the given set of properties `props`. 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. In particular, `local-cflow` and `local-cflowbelow` inspect the call stack of join points as long as they are on the same host.

5.3. 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 scoping strategies as simple transformers of non-distributed strategies, as will be shown in Section 6.

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

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. 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. These extensions enable us to define distribution related pointcut designators. For instance, in Fig. 10, the function `remote?` discriminates remote join points from local ones. Similarly, `remote-call` and `remote-copy` discriminate the different types of join points.

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

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

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

Fig. 10. More distribution-related pointcut designators.

6. Scoping of distributed aspects

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 [19], AWED [3] and ReflexD [29]. 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, 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.

Based on an analysis of what is missing in a distributed setting, this section first sketches our proposal for expressive scoping of distributed aspects, and shows how we can express the previous cases succinctly. We then make our proposal precise by revisiting the semantics of Section 4 to take into account the extended execution model of Section 5. Our definitional interpreter is updated accordingly. Finally, we define *distributed* scoping strategies as transformers of scoping strategies, succinctly expressing the solutions to the examples given in Section 2.4.

6.1. Analysis of the problem

The requirements of the distributed deployment scenarios of Section 2.4 suggest that, in addition to specifying a scoping 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(s) 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].

6.2. Distributed scoping strategies

How shall we add the above-mentioned properties to plain scoping strategies? The basic idea is to define distributed scoping strategies as transformers over plain scoping strategies: an application $t(\sigma)$ of a strategy transformer t to a (non-distributed) scoping strategy σ augments the latter by a specification of distributed propagation and/or activation. In this way, scoping 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 scoping strategies:

1. *propagate-global*: always propagate.
2. *propagate-local*: never propagate to other hosts.
3. *propagate-remote*: only propagate in remote calls.
4. *active-global*: always active.
5. *active-local*: never active in other hosts.
6. *active-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 scoping strategies above can in turn be defined in terms of two general distributed strategies *propagate-if* and *active-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-if* (*hp*): propagate to hosts matched by *hp*.
- *active-if* (*hp*): active on hosts matched by *hp*.

6.3. 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 scoping strategies are functions that take other strategies as parameters, we assume strategies to be first-class values in the language. We use the bracketed syntax <. . .> to construct scoping strategies as values. For instance, the dynamic scoping strategy found in both CaesarJ and AspectScheme (call stack propagation only, no filter, see 4.3) is: `dynamic = <f-true, f-false, f-true>`, where `f-true` (resp. `f-false`) is the constant function that returns true (resp. false).

Case 1. With scoping strategies, the simple aspect introduced in Section 2.4 can be directly used:

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

Its pointcut is not cluttered with scoping information, which is expressed when the apsect is deployed:

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

Booking booking = null;
deploy [(propagate-if(inTravelGroup))(notBeyondDB)](new Billing()){
  booking = confirm(res);
}
```

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 the plain scoping strategy `notBeyondDB`. Here, `target` is similar to the AspectJ pointcut designator `target`, selecting join points based on the type of the target object.

The scenario also stipulates that the aspect should only propagate in hosts that belong to the `TravelGroup`. In order to do so, the strategy transformer `propagate-if(inTravelGroup)` is applied to the strategy `notBeyondDB`, where the predicate `inTravelGroup` selects the appropriate hosts.

If the aspect had to be deployed on all hosts, a *propagate-global* strategy could have been used:

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

Note that the `propagate-global` transformer is optional because it is applied by default.

Case 2. Again, a simple profiling aspect can be used:

```
aspect Profiling {
  pointcut Profile(): execution(* *.*(..));

  before(): profile() { [...] }
  after(): profile() { [...] }
}
```

This aspect should be attached to the client, propagated globally (the default), but only *active* locally. This corresponds to the use of a `deploy-on` construct with an *active-local* strategy:

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

where `dynamic` refers to the dynamic scoping strategy like in AspectScheme and CaesarJ, as defined above.

$$A_{call} = \{\langle a, \sigma\rangle \in A \mid \sigma = \langle c, d, f\rangle \wedge c(njp)\} \quad (1)$$
$$A_{exec} = A_{call} \cup closure.A \quad (2)$$
$$A_{new} = \{\langle a, \sigma\rangle \in A \mid \sigma = \langle c, d, f\rangle \wedge d(njp)\} \quad (3)$$
$$A_{copy} = \{\langle a, \sigma\rangle \in A \cup original.A \mid \sigma = \langle c, d, f\rangle \wedge d(njp)\} \quad (4)$$

Fig. 11. Revisiting scoping strategies semantics.

Case 3. As in the previous cases, the simple privacy aspect introduced in Section 2.4 can be used:

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

In this case, without considering distribution, the scoping strategy is a refinement of statically-scoped aspects *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. The deployment of the aspect is expressed as follows:

```
Strategy inTravelers = <f-false, target(Traveler), f-true>;
TravelerFactory factory = ...
Privacy privacy = new Privacy();

deploy-on[active-remote(inTravelers)](privacy, factory);
```

The construct `deploy-on` deploys the aspect within the factory object. Actually, the scenario stipulates that the aspect is only active in hosts that do not belong to the `TravelGroup`. The scoping strategy has to be refined as follows:

```
deploy-on[(active-if(not(inTravelGroup)))(inTravelers)]
  (privacy, factory);
```

To summarize, all the examples can be expressed by augmenting the plain scoping 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 section details the semantics of our proposal, by further extending our Scheme interpreter.

6.4. Refining scoping strategies

Our analysis of Section 6.1 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 scoping strategies by introducing the two refinements to the execution model introduced in Section 5.3. 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 scoping strategies [25] however only considers function application join points. Therefore the semantics of scoping strategies as presented in Fig. 7 needs to be revisited. The new semantics is presented on Fig. 11 (note that A_{weave} is unchanged).

First, to account for the separation of call and execution join points, we update the semantics of scoping strategies to separate A_{app} into two aspect environments: A_{call}, the set of aspects that propagate on the call stack (1), and A_{exec}, the set of aspects to use during the evaluation of the body of a function (2). 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} (3) and distinguish a new aspect environment, A_{copy}, which is the set of aspects that are embedded in a function copy (4). 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 decides 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.

In the following section we present the corresponding interpreter. Section 6.6 then shows that armed with these new definitions, combined with the definition of join points augmented for distribution (with current and target hosts), distributed scoping strategies can be expressed simply as scoping strategy transformers.

6.5. Interpretation

We now describe the semantics of our updated model of scoping strategies using an definitional interpreter. The interpreter of Fig. 12 extends the interpreter of the higher-order distributed language (Fig. 8) with the scoping strategy feature presented in Section 4.

```
(define (eval exp E A jp)      ⟵ 1
  (cond ...
        ((app? exp)
         (let ((cl    (eval (app-fun exp) E A jp))
                      (args (eval-args (app-args exp) E A jp)))
           (eval-app cl args A jp))))
        ...))

(define (eval-app cl args A jp)
  (let ((njp (make-jp 'call cl args jp (get-current-host)
                      (target-host cl))))      ⟵ 2
    (weave-some A njp)      ⟵ 3
    (let ((asps (collect-match-c njp A)))      ⟵ 4
      (if (fun-stub? cl)
        (remote-exec cl args asps jp)
        (eval-exec   cl args asps jp)))))

(define (eval-exec cl args A jp)
  (let ((njp  (make-jp 'exec cl args jp
                       (get-current-host) #f)))     ⟵ 5
        (env  (extend-env (closure-params cl) args (closure-env cl)))
        (asps (union A (closure-aspects cl)))      ⟵ 6
    (weave-some A njp)      ⟵ 7
    (eval (closure-body cl) env asps njp)))

(define (new-function formals body E A jp)
  (let* ((njp  (make-jp 'new #f formals jp
                        (get-current-host) #f))      ⟵ 8
         (asps (collect-match-d njp A)))      ⟵ 9
    (weave-some A njp)      ⟵ 10
    (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)))      ⟵ 11
         (asps (union (collect-match-d njp A)
                      (collect-match-d njp (closure-aspects orig)))))     ⟵ 12
    (weave-some A njp)      ⟵ 13
    (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))      ⟵ 14
```

Fig. 12. Interpretation of scoping strategies for a higher-order distributed procedural language.

First of all, note that `eval` takes as parameter the current aspect environment, in addition to the lexical environment and the current join point 1. When a function is applied (`eval-app`), first the corresponding call join point is created 2. 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 3: `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 14.

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` 4. If the function is a stub, the remote call execution is triggered as explained in Section 5.1, 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 5 and woven 7. The set of aspects that potentially apply during the evaluation of the function body, A_{exec}, is computed 6 and evaluation proceeds.

Similarly, when a function is defined (`new-function`), the corresponding new join point is created 8 and woven 10. 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` 9.

When a function is copied (`copy-function`), a copy join point is created 11. 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 [13], 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 [12].

6.6. Distributed scoping strategies

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

Distributed scoping strategies are transformers that take a scoping strategy as parameter 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 5.2.

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 Fig. 9, 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 scoping strategy for propagation, `propagate-if` (Section 6), as follows:

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

(For the sake of conciseness, we reuse the bracketed syntax introduced in Table 1 to do pattern matching on scoping strategies as well as to build these strategies.) Given a host predicate, `propagate-if` returns a strategy transformer that, given a scoping strategy $\sigma = \langle \mathtt{c}, \mathtt{d}, \mathtt{f} \rangle$, returns a new strategy σ' where the propagation components `c` and `d` are limited by 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 scoping 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 scoping strategy for activation, `active-if` (Section 6), 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 scoping strategy $\sigma = \langle \mathtt{c}, \mathtt{d}, \mathtt{f} \rangle$, returns a new strategy σ' in which the join point filter `f` is complemented by a host restriction. The propagation components `c` and `d` are unchanged.

Typical scoping strategies. The six simple scoping strategies introduced in Section 6.2 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.[10]

Finer-grained strategies. While distributed scoping strategies are assimilating the two propagation dimensions of scoping 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.

7. The scoping scenarios revisited

We now show succinctly how the scoping scenarios defined in Section 2.4 and implemented in Section 3 using AspectJ, CaesarJ and AWED can be implemented using our approach. We also revisit the limiting assumptions (**A1**), (**A2**), (**A3**) of Section 3.1 and Section 3.3, and (**B1**), (**B2**) of Section 3.2 to establish if our approach also requires them to be made.

7.1. Scoping scenarios implemented

In Section 6.3 we have expressed the example cases in a Java-like syntax. Here we repeat these deployment scenarios, using our Scheme-like language. Recall that in the following, `dynamic` is the dynamic scoping strategy found in both CaesarJ and AspectScheme (call stack propagation only, no filter, see 4.3) and is defined as `<f-true, f-false, f-true>`, where `f-true` (resp. `f-false`) is the constant function that returns true (resp. false). We make use of these constant functions in different places below.

Case 1: Controlling Propagation. The billing aspect is deployed over a dynamic extent, bound on calls to the database access, which is performed by calling the `dbaccess` function. Furthermore, the aspect should only propagate in hosts that belong to the `TravelGroup`, which is established by a `inTravelGroup` predicate. Deployment is specified as follows:

```
(let ((notBeyondDB <! (call dbaccess), f-false, f-true>))
  (deploy (propagate-if inTravelGroup notBeyondDB) billing
    (confirm reservation)))
```

The first line establishes the deployment strategy for the bounded dynamic scope, which is then used by the `propagate-if` strategy transformer.

Case 2: Controlling Activation. The profiling aspect should propagate globally but be only active locally. This is specified using the dynamic scoping strategy `dynamic`, transformed by the `active-local` strategy transformer:

```
(deploy-on (active-local dynamic) profiling client)
```

Case 3: Controlling Per-Object Activation. The scoping strategy used embeds the aspect in all objects created by the `make-traveler` factory (playing the role of class instantiation), while the strategy transformer ensures that the aspect is active when not inside the `TravelGroup`.

```
(let ((inTravelers  <f, (call make-traveler), f-true))
  (deploy-on (active-if (not inTravelGroup) inTravelers) privacy factory))
```

7.2. Implementation assumptions revisited

In Section 3 we needed to make three limiting assumptions (**A1**), (**A1**), (**A3**) when attempting the implementation with AspectJ, two assumptions with CaesarJ (**B1**), (**B2**) and for AWED the assumption (**A1**) needed to be taken. We revisit these assumptions and assert whether these need to be made when using our approach.

> **A1:** We need to be able to weave all affected classes and update them in a consistent way across the entire distributed system.

In our approach aspects are *dynamically* deployed, and properly follow both execution flows and objects as required, without having to prepare the involved sites in any way. Therefore this assumption does not need to be made.

[10] Here, it is particularly important to use a curried version of `eq?` because, as is typical in distributed systems, `current-host` is a dynamically-scoped variable. In `is-current`, we need however to capture the host bound to `current-host` where it is defined, not where it is used; currification ensures this.

A2: For all method executions in the distributed control flow that we wish to capture, there is at least one parameter `obj` that exists at the beginning of the control flow and is reachable at that point.

A3: (A copy of) the above parameter `obj` is not stored within this control flow and later retrieved, nor is it concurrently accessed.

Both these assumptions were required because AspectJ does not provide support for distributed control flow. As our approach *does* provide support for distributed control flow, these assumptions do not need to be made.

B1: The weather server is the only server outside of the travel group.

In CaesarJ specific host information cannot be tested at the pointcut level, and host information is not available to the advice. Eliminating propagation outside of the travel group needs to be done by a `cflow` test verifying calls to the weather server. In our approach host information *can* be tested by the pointcut, so this assumption does not need to be made.

B2: The travel agent client is not multi-threaded and does not perform other computations than the ones we are interested in.

This assumption needed to be made because in CaesarJ aspects cannot be deactivated along a distributed control flow. In our approach this *is* possible, as a result this assumption does not need to be made.

To sum up, our approach effectively addresses the issues raised with respect to other proposals, because none of the stated assumptions need to be made. We elaborate on a specific feature that is particularly relevant, runtime weaving, in Section 8.2 below.

8. Discussion

We now briefly discuss the major limitations of our approach, and the perspectives they open. We also report on what lessons can be learned from this experiment with respect to implementations of aspect weaving.

8.1. Limitations and perspectives

This work introduces a general notion of aspect scoping for distributed programming. Our model includes first-class pointcuts and advices, and as such, completely supports applying aspects to other aspects. The model deals with the potentially remote execution of both pointcuts and advices, 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 memorizes results of remote invocations.

8.2. On the necessity of runtime weaving

Beyond the proposal of scoping strategies, this work reports on a reasonably extensive case study with distributed aspect-oriented programming. Section 3 describes various attempts at implementing advanced scoping scenarios with AspectJ and RMI, CaesarJ, and AWED. From the issues we encountered with these systems, a clear lesson can be drawn with respect to the implementation strategy of aspects.

Folklore has it that aspect weaving is a compile-time process. While clearly incorrect from a conceptual point of view, this belief reflects the fact that AspectJ, and many AOP languages, are implemented using (source or byte) code transformation, prior to runtime. Code transformation consists of merging (the statically-determinable bits of) aspects with the application code. This implementation strategy has the merit of being efficient. It is, however, less expressive than seeing weaving as a runtime process, because some ties between aspects and base code cannot easily be undone dynamically. In a distributed setting, this means that aspects get woven and hardwired into serialized data, leading to the versioning issues we have reported. Dynamic weaving has received a lot of attention as well, especially from the research community, and it has been shown that aspect-aware execution environments can be both flexible and efficient [5–7,21].

From the experience reported in this paper, a strong limitation of compile-time weaving is made evident: if done statically, aspect weaving interferes with *class versioning*. The issue of maintaining a consistent code base in a distributed

system is already far from trivial in a non-AOP setting. If aspect weaving affects base code, the consistency issue appears each time a different set of aspects is used in one of the hosts of the distributed system. In contrast, by maintaining aspects separate from base code and leaving the responsibility of weaving to the runtime, the code consistency issue is not affected. Furthermore, runtime weaving enables the manipulation of aspect environments as described in this paper. Therefore, both code consistency in the presence of versioning and expressive scoping can be achieved through runtime weaving.

Finally, the coordinated and flexible application of aspects to different parts of a distributed systems is very difficult to handle using static aspect weaving but also far from trivial if dynamic aspect weaving is used: Truyen and Joosen [30] have proposed a notion of atomic weaving of dynamic aspects that requires substantial non-standard infrastructure support; the AWED system has been extended by causal relationships between events to ensure coordinated aspect application [2]. Distributed scoping strategies as introduced in this paper provide a different, promising approach to this problem: consistent deployment is facilitated by decoupling the deployment proper by means of an expressive deployment language from the aspect implementation itself that is done taking a view local to where it is initially deployed.

9. 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 scoping strategies [25,26] to deal with the distribution dimension of scoping. In the line of previous research, as well as the Aspect SandBox project [16,17,31], we have given the operational semantics of our proposal as a concise Scheme interpreter.

Distributed scoping strategies provide precise control over the two locality dimensions of aspects in distributed systems: propagation and activation. To achieve this, previous work on scoping 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 scoping strategies as transformers of plain scoping strategies. Because scoping 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 dynamically-deployed aspects in distributed systems 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.

References

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, Klaus Ostermann, An overview of CaesarJ, in: Transactions on Aspect-Oriented Software Development, in: Lecture Notes in Computer Science, vol. 3880, Springer-Verlag, 2006, pp. 135–173.
[2] Luis Daniel Benavides Navarro, Rémi Douence, Mario Südholt, Debugging and testing middleware with aspect-based control-flow and causal patterns, in: Proceedings of the 9th ACM/IFIP/USENIX International Middleware Conference, Leuven, Belgium, Springer-Verlag, 2008.
[3] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, Davy Suvée, Explicitly distributed AOP using AWED, in: Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, ACM Press, 2006, pp. 51–62.
[4] Lodewijk Bergmans, Mehmet Akşit, Composing crosscutting concerns using composition filters, Communications of the ACM 44 (10) (2001) 51–57.
[5] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, Mira Mezini, Adapting virtual machine techniques for seamless aspect support, in: OOPSLA 2006 [20], ACM SIGPLAN Notices, 41(10), pp. 109–124.
[6] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, Mira Mezini, Efficient control flow quantification, in: OOPSLA 2006 [20], ACM SIGPLAN Notices, 41(10), pp. 125–138.
[7] Christoph Bockish, Michael Haupt, Mira Mezini, Klaus Ostermann, Virtual machine support for dynamic join points, in: Lieberherr [15], pp. 83–92.
[8] Bruno De Fraine, Mathieu Braem, Requirements for reusable aspect deployment, in: Markus Lumpe, Wim Vanderperren (Eds.), Proceedings of the 6th International Symposium on Software Composition, SC 2007, Braga, Portugal, in: Lecture Notes in Computer Science, vol. 4829, Springer-Verlag, 2007.
[9] Christopher Dutchyn, David B. Tucker, Shriram Krishnamurthi, Semantics and scoping of aspects in higher-order languages, Science of Computer Programming 63 (3) (2006) 207–239.
[10] Matthias Felleisen, On the expressive power of programming languages, Science of Computer Programming 17 (1991) 35–75.
[11] Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes, Essentials of Programming Languages, 2nd ed., The MIT Press, 2001.
[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William Griswold, An overview of AspectJ, in: Jorgen L. Knudsen (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP 2001, in: Lecture Notes in Computer Science, vol. 2072, Springer-Verlag, Budapest, Hungary, 2001, pp. 327–353.
[13] Ramnivas Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Press, 2003.
[14] Bert Lagaisse, Wouter Joosen, True and transparent distributed composition of aspect-components, in: Maarten van Steen, Michi Henning (Eds.), Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference, Middleware 2006, in: Lecture Notes in Computer Science, vol. 4290, Springer-Verlag, Melbourne, Australia, 2006, pp. 42–61.
[15] Karl Lieberherr (Ed.), Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, ACM Press, 2004.
[16] Hidehiko Masuhara, Gregor Kiczales, Modeling crosscutting in aspect-oriented mechanisms, in: Luca Cardelli (Ed.), Proceedings of the 17th European Conference on Object-Oriented Programming, ECOOP 2003, in: Lecture Notes in Computer Science, vol. 2743, Springer-Verlag, Darmstadt, Germany, 2003, pp. 2–28.

[17] Hidehiko Masuhara, Gregor Kiczales, Christopher Dutchyn, A compilation and optimization model for aspect-oriented programs, in: G. Hedin (Ed.), Proceedings of Compiler Construction, CC2003, in: Lecture Notes in Computer Science, vol. 2622, Springer-Verlag, 2003, pp. 46–60.
[18] Mira Mezini, Klaus Ostermann, Object creation aspects with flexible aspect deployment. Technical report, Technische Universität Darmstadt, 2003.
[19] Muga Nishizawa, Shigeru Chiba, Michiaki Tatsubori, Remote pointcut — a language construct for distributed AOP. In Lieberherr [15], pp. 7–15.
[20] Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2006, Portland, Oregon, USA, October 2006. ACM Press. ACM SIGPLAN Notices, 41(10).
[21] Andrei Popovici, Gustavo Alonso, Thomas Gross, Just-in-time aspects: efficient dynamic weaving for Java, in: Mehmet Akşit (Ed.), Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development, AOSD 2003, Boston, MA, USA, ACM Press, 2003, pp. 100–109.
[22] Hridesh Rajan, Kevin Sullivan, Eos: Instance-level aspects for integrated system design, in: Proceedings of ESEC/FSE 2003, Helsinki, Finland, 2003, pp. 297–306.
[23] SUN Microsystems, Remote Method Invocation, 1998.
[24] Éric Tanter, Controlling aspect reentrancy, Journal of Universal Computer Science 14 (21) (2008) 3498–3516.
[25] Éric Tanter, Expressive scoping of dynamically-deployed aspects, in: Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development, AOSD 2008, Brussels, Belgium, ACM Press, 2008, pp. 168–179.
[26] Éric Tanter, Beyond static and dynamic scope, in: Proceedings of the 5th ACM Dynamic Languages Symposium, DLS 2009, Orlando, FL, USA, ACM Press, 2009, pp. 3–14.
[27] Éric Tanter, Higher-order aspects in order, in: Scheme and Functional Programming Workshop, Boston, MA, USA, 2009.
[28] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, Mario Südholt, Expressive scoping of distributed aspects, in: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, ACM Press, 2009, pp. 27–38.
[29] Éric Tanter, Rodolfo Toledo, A versatile kernel for distributed AOP, in: Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems, DAIS 2006, in: Lecture Notes in Computer Science, vol. 4025, Springer-Verlag, Bologna, Italy, 2006, pp. 316–331.
[30] Eddy Truyen, Wouter Joosen, Run-time and atomic weaving of distributed aspects, Transactions on Aspect-Oriented Software Development II 4242 (2006) 147–181.
[31] Naoyasu Ubayashi, Genki Moriyama, Hidehiko Masuhara, Tetsuo Tamai, A parameterized interpreter for modeling different AOP mechanisms, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, CA, USA, ACM Press, 2005, pp. 194–203.
[32] Mitchell Wand, Gregor Kiczales, Christopher Dutchyn, A semantics for advice and dynamic join points in aspect-oriented programming, ACM Transactions on Programming Languages and Systems 26 (5) (2004) 890–910.