Taming Aspects with Membranes

Éric Tanter, Nicolas Tabareau, Rémi Douence

Taming Aspects with Membranes

Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile — Chile
etanter@dcc.uchile.cl

Nicolas Tabareau Rémi Douence
ASCOLA group
INRIA
Nantes — France
first.last@inria.fr

Abstract

In most aspect-oriented languages, aspects have an unrestricted global view of computation. Several approaches for aspect scoping and more strongly encapsulated modules have been formulated to restrict this controversial power of aspects. This paper proposes to leverage the concept of *programmable membranes* developed by Boudol, Schmitt and Stefani, as a means to tame aspects by customizing the semantics of aspect weaving locally. Membranes subsume previous proposals in a uniform framework. Because membranes give structure to computation, they enable flexible scoping of aspects; because they are programmable, they make it possible to define visibility and safety constraints, both for the advised program and for the aspects. We first describe membranes for AOP without committing to any specific language design. In addition, we then illustrate an extension of AspectScheme with membranes, and explore the instantiation of programmable membranes in the Kell calculus. The power and simplicity of membranes open interesting perspectives to unify multiple approaches that tackle the unrestricted power of aspect-oriented programming.

Keywords Aspect-oriented programming, aspect scoping, programmable membranes

1. Introduction

In the pointcut-advice model of aspect-oriented programming, crosscutting behavior is defined by means of pointcuts and advices. Because join points identified by pointcuts can be scattered, weaving typically requires aspects to have a global view of computation. The fact that aspects have an unrestricted global view on computation has however raised many concerns about the pertinence of AOP, especially in terms of modular reasoning [1]. Different proposals have emerged to tackle this extreme power given to aspects. On the one hand, some proposals make it possible to control the *scope* of aspects; the scope of an aspect is defined as the set of join points the aspect *sees*, *i.e.* against which its pointcuts are matched. Examples include statically and dynamically-scoped aspects [7], scoping strategies [15], as well as execution levels [17]. These proposals can be seen as ways to reduce the impact of an aspect on the aspect deployment side. On the other hand, other proposals have adopted the dual perspective and focused on how to make it possible for a given module to *protect* its own computation from advising. Examples include Open Modules [1], XPIs [8], and IIIA [14]. Yet other proposals rely on types to control the *effects* that aspects can induce [10].

Stepping back, aspect-oriented programming can be seen as exposing computation as events to which aspects can react. The underlying issues discussed above are therefore related to scoping and control of events in general. These issues have been explored in particular by the distributed systems community (where, of course, many other concerns apply). In this respect, we find the notion of *programmable membranes* developed by Boudol [5] and Schmitt and Stefani [12], themselves loosely inspired by the biological notion of cells and membranes, to be particularly appealing as a general control mechanism. This paper proposes to adapt the notion of programmable membranes to control the controversial power of aspects. We describe a notion of membranes in an AOP context, which subsumes and makes it possible to combine various existing proposals for controlling aspects, in a single uniform framework.

To introduce our proposal, let us first consider Figure 1, which depicts the basic weaving protocol between a base program and an aspect. The program emits join points that are passed to the aspect for weaving. The aspect may then proceed (with a possibly modified join point). When the program is done with the original computation, the value is passed back to the aspect so that it can complete its advice. Finally, the advice returns a (possibly modified) value, which is used to resume the base program.

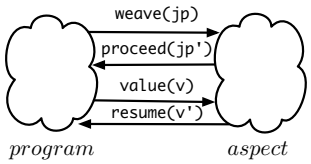

Figure 1. Aspect weaving protocol.

The basic idea of our proposal is to wrap the program and the aspect of Figure 1 inside their own *membranes*. In our approach, membranes are thus overlaid on top of computation, in charge of propagating joint points and controlling the weaving protocol of their inner computation. We introduce the possibility to register aspects in membranes, and to bind membranes so as to advise the computation of other membranes. Because membranes are programmable, join point propagation and weaving can be customized locally.

Programmable membranes bring two major benefits to AOP: *(a)* because they give structure to computation, they enable flexible scoping (they actually generalize the notion of execution levels [17] to arbitrary topologies); *(b)* because they are programmable, they make it possible to define visibility and safety constraints on both the "base" and "aspect" sides of the weaving protocol, hence tailoring particular weaving semantics and guarantees *locally*.

This paper first explores the notion of programmable membranes for AOP, describing the general concepts and benefits of membranes for taming aspects—without committing to a specific language design. Still, beyond this conceptual contribution, this paper contributes both a concrete and a formal instantiation of membranes for AOP.

After a brief introduction to execution levels, highlighting their topological limitations (Section 2), we give a high-level overview of membranes for AOP, describing their application to control visibility of join points and enforce safety and encapsulation properties (Section 3). Section 4 describes the benefits of membranes derived from flexible topological scoping, which completely subsumes execution levels [17]. Section 5 goes further by describing different relations between membranes (co-observation, hierarchical nesting, and crosscutting membranes) and their applications. Once the different concepts and benefits of the proposals are presented, we progressively dive into more details. Section 6 clarifies the relation between membranes and computation, and discusses a range of possible instantiations of the model for concrete membrane-based AOP languages. We illustrate a specific point in the design space of membrane-based AOP languages with MAScheme, an extension of AspectScheme [7] with membranes, showing how different examples can be expressed. In Section 8, we explore the instantiation of membranes for AOP in the Kell calculus [12], describing different kinds of programmable membranes. Finally, Section 9 puts membranes in perspectives and Section 10 concludes.

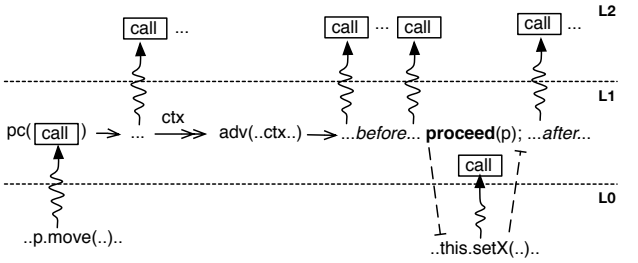

Figure 2. Execution levels in action: pointcut and advice are evaluated at level 1, *proceed* goes back to level 0 (from [17]).

2. Execution Levels: Benefits and Limits

An aspect observes the execution of a program through its pointcuts, and affects it with its advice. An advice is a piece of code, and therefore its execution also produces join points. Similarly, pointcuts as well can produce join points. For instance, in AspectJ, one can use an *if* pointcut designator to specify an arbitrary Java expression that ought to be true for the pointcut to match. The evaluation of this expression is a computation that produces join points. In higher-order aspect languages like AspectScheme [7] and others, all pointcuts and advice are standard functions, whose application and evaluation produce join points as well.

The fact that aspectual computation produces join points raises the crucial issue of the *visibility* of these join points. In most languages, aspectual computation is visible to all aspects—including themselves. This of course opens the door to infinite regression and unwanted interference between aspects. These issues are typically addressed with ad-hoc checks (*e.g.,* using *!within* and *cflow* checks in AspectJ) or primitive mechanisms (like AspectScheme's *app/prim*). However, all these approaches eventually fall short for they fail to address the fundamental problem, which is that of *conflating* levels that ought to be kept separate [6].

2.1 Execution levels in a nutshell

In order to address the above issue, Tanter proposed execution levels for AOP [17]. A program computation is structured in *levels*. Computation happening at level 0 produces join points observable at level 1 only. Aspects are *deployed* at a particular level, and observe only join points at that level. This means that an aspect deployed at level 1 only observes join points produced by level-0 computation. In turn, the computation of an aspect (*i.e.* the evaluation of its pointcuts and advice) is reified as join points visible at the level immediately above: therefore, the activity of an aspect standing at level 1 produces join points at level 2.

An aspect that acts *around* a join point can invoke the original computation; in AspectJ, this is done by invoking *proceed* in the advice. The original computation ought to

run at the same level at which it originated![1] In order to address this issue, it is important to remember that when several aspects match the same join point, the corresponding advices are chained, such that calling *proceed* in advice k triggers advice $k + 1$. Therefore, the semantics of execution levels guarantees that the *last call* to *proceed* in a chain of advice triggers the original computation at the lower original level.

This is shown in Figure 2. A call to a *move* method in the program produces a call join point (at level 1), against which a pointcut *pc* is evaluated. The evaluation of *pc* produces join points at level 2. If the pointcut matches, it passes context information *ctx* to the advice. Advice execution produces join points at level 2, except for *proceed*: control goes back to level 0 to perform the original computation, then goes back to level 1 for the after part of the advice.

2.2 Benefits and limits

By separating execution into levels, unwanted interactions between aspects are avoided. For instance, it becomes possible to reuse off-the-shelf dynamic analysis aspects and apply them to a given program, with consistent semantics [18]. It is also possible to apply aspects to other aspects, by deploying aspects at higher levels (for instance, detecting data races in a profiling aspect running at level 1 with an instance of Racer [4] running at level 2).

The structure that execution levels bring to computation is crucial to properly scope aspects and therefore limit their "global view" of a program computation. However, because execution levels are directly inspired by work on the reflective tower [13], they represent but one possible topology for computation: that of a one-dimensional tower. Such a tower is actually quite restrictive. For instance, it is not possible to have an aspect at level 2 observe only the computation of one aspect at level 1, without seeing at the same time all computation that happens at level 1. Similarly, if an aspect needs to observe computation at both levels 0 and 1, because execution levels are not "transparent" (an aspect only sees computation at the level immediately below it), the only possibility is to deploy the aspect twice, at levels 1 and 2; but this raises the possibility for the aspect to see its own computation again, thereby defeating one of the main motivation for execution levels.

To sum up, giving structure to computation is important in aspect-oriented programming; but there is no reason to stick to a rigid topology like that of execution levels. This paper subsumes execution levels by exploring a much more flexible structuring mechanism: membranes.

3. Programmable Membranes

We propose to adapt the notion of programmable membranes [5, 12] as a means to structure execution and control aspect weaving: membranes can be deployed around a given computation and serve as a scoping mechanism for the join points emitted by that computation. A membrane is itself a programmable entity that is responsible for a number of decisions, such as dealing with the propagation of join points produced by its inner computation.

[1] This issue is precisely why using control flow checks in AspectJ in order to discriminate advice computation is actually flawed [17].

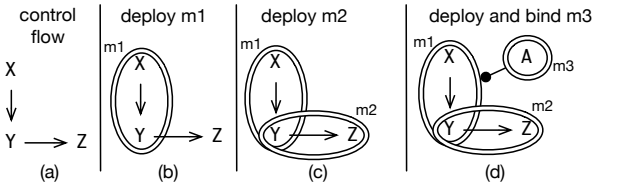

Figure 3. Membranes deployment and aspectual bindings. Membrane m1 (resp. m2) wraps computation X and Y (resp. Y and Z). Membrane m3 is bound to m1: join points from m1's computation will be visible to m3's aspect, A.

3.1 Deploying and binding membranes

In the general case, membranes can be deployed around any computation. Consider for instance a control flow graph between arbitrary computations X, Y and Z (Fig. 3(a)). One can deploy a membrane m1 around the computations X and Y (b); and a membrane m2 around Y and Z (c). Membranes control propagation of join points produced by these computations. Membranes give structure to the computation so that aspects can be flexibly scoped. Aspects can be *registered* in a membrane. We call *advising membrane* a membrane that is *bound* to another membrane, called *advised membrane*. Aspects registered in an advising membranes are woven on the join points emitted by advised membranes. In our example, aspect A is registered in membrane m3; m3 is bound to m1 (d), as denoted by the lollypop arrow. As a result, join points produced by the computation of X and Y will be visible to A. The advising relation between membranes derived from binding opens the door for topological scoping of aspects (Sections 4 and 5).

3.2 Propagation of join points and aspect weaving

Binding m3 to m1 informs m1 that m3 wishes to see (and potentially affect) the join points produced by the computation inside m1. For each produced join point, m1 is then free to decide whether to propagate the join point to m3 or not (and also to enforce certain restrictions, as will be discussed later). Figure 4 illustrates join point propagation and aspect weaving with membranes. Computation X produces a join point jp (1); this join point is then absorbed by the membrane m1 (2). Since m1 is a programmable membrane, it can implement different visibility policies. Suppose that m1 decides to propagate the join point to m3, it will then make the join available in its outer environment, tagging it with the destination membrane m3 (3). On its side, m3 always listens for join points addressed to it in its outer environment. When

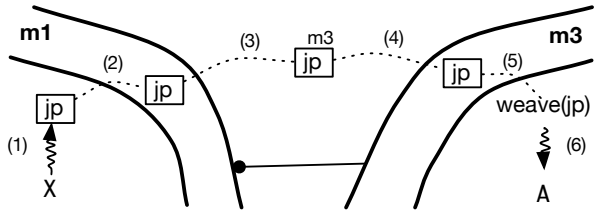

Figure 4. From join point emission to aspect weaving. Computation X produces a join point (1), which travels through membranes m1 and m3 (2-5) before being processed by aspect A (6).

a join point is addressed to it, it is absorbed in the membrane m3 (4). Here again, because m3 is programmable, it may decide to discard the join point, or to actually weave its inner aspects (5); in that case, aspects residing inside m3 are woven on the join point (6).

3.3 Visibility and safety policies

Binding membranes together produces particular topological arrangements, which can be used for flexible scoping of aspects; this is explored further in Sections 4 and 5. In addition to topological scoping, programmable membranes can be used to control at a fine-grained level the propagation of join points between membranes. *Transparent membranes* simply relay join points that are either emitted by their inner computation or received (for advising) from their outer environment. For either security or encapsulation reasons, it is however sometimes necessary to protect some computation from aspect advising. This can be achieved with membranes by defining *opaque membranes*, *i.e.* that never let any join point traverse through them. In between transparent and opaque membranes, *translucent* membranes relay only a subset of the join points produced by their inner computation. This makes it possible to use membranes to model Open Modules [1]. The model also allows for opacity to be dynamic; a given membrane can transition from transparent to translucent or opaque based on characteristics of its execution environment.

Programmable membranes can also be used to enforce safety policies that limit the power of aspects, as in [10], for instance by preventing aspects from changing the arguments of proceed, or its return value. To see how this can be achieved, it is necessary to detail a bit more the weaving process described above (Figure 4). When aspect A is woven on join point jp, it executes its before advice and then informs m1 that it wants to proceed (potentially with a different join point `jp`$'$) by sending a message. The message flows from m3 to m1, and then m1 invokes the original computation X. The base result v is then repropagated back to m3 so that A can execute its after advice. Finally, when A finishes, it tells m1 to resume (potentially with a different result v') .

Therefore, a membrane can simply store the original join point jp and use it (instead of the potentially modified jp') in order to trigger the original computation. This means that any modification to the join point (*e.g.,* new arguments) will not be considered. Similarly, the return value of the original computation can also be stored and used when resuming the base computation after weaving has finished. Another example of a safety policy is to program the membrane such that there is exactly one proceed call to the original computation. If an aspect tries to perform more than one proceed, the membrane can skip it and return the result of the previous proceed or a default value. It can also insert a call to proceed when no aspect call it (for instance with the original arguments, their current values if they have been modified by the advice, or default values). An *immutable membrane* is a membrane that enforces all the properties described above: exactly one call to proceed, with a fixed join point, and a fixed return value.

For instance, let us consider a web browser executed in one membrane and a caching aspect running in another membrane bound to the browser membrane. When the browser wishes to download a URL with `get(URL)`, the corresponding join point is propagated to the aspect that checks if the corresponding page is already stored in its cache. A translucent membrane around the browser computation can filter out HTTPS requests so that they are not visible to the caching aspect. If the browser membrane is made immutable, then we are sure that even an untrusted cache aspect cannot create security leaks, *e.g.,* by changing the actual URL being retrieved.

Variations are endless. For instance, a translucent membrane can anonymize outgoing join points by erasing or obfuscating information (*e.g.,* login or password parameters). A membrane can also adapt parameters of incoming join points before they are passed to its registered aspects.

3.4 Symmetry of the model

Membranes can play a dual role, as both advised and advising membranes. On the one hand, they control the emission of join points from their inner computation towards advising membranes. This is useful to control if and how specific join points are propagated, to which advising membranes and in which order, and possibly implementing safety policies as described above. On the other hand, membranes receive join points for weaving by their registered aspects, thereby controlling which join points are actually seen, and possibly controlling the weaving order of its registered aspects.

The model is therefore totally symmetric. Both roles of a membrane can be programmed independently of each other, or cooperate if required. This is particularly important when talking about transparent/translucid/opaque membranes. Indeed, a membrane can be programmed with a translucent advised interface *and* a transparent advising interface. As a consequence, its aspects see all join points that it receives,

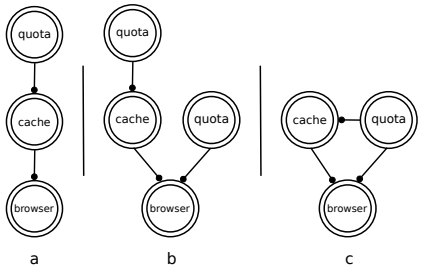

Figure 5. Topological scoping. (a) tower—corresponds to execution levels; (b) tree; (c) DAG.

and it can control which join points of its inner computation are visible to the outside.

4. Basic Topological Scoping

In addition to enabling safety properties, membranes make it possible to express flexible *topologies* of computation, going beyond execution levels [17]. This section shows how basic topological scoping can be achieved. We illustrate the flexibility of the model by first describing how to express the execution levels (Section 4.1). We then go beyond levels by describing tree-based (Section 4.2) and DAG-based topologies (Section 4.3).

4.1 Membranes in a tower: execution levels

The basic example of a browser and a caching aspect, each running in their own membranes, has a two-level topology: the browser running at level 0, and the caching aspect at level 1. Having more aspects at level 1 could be done by registering them in the same membrane with the caching aspect. There is a one-to-one correspondence between execution levels and membranes stacked on top of each other (one membrane per level), each membrane advising the one below.

For instance, suppose that the cache aspect stores the pages in a file and that we wish to limit the disk consumption of this aspect. We can simply introduce a quota aspect that applies to the cache aspect: in other words, we can register the cache aspect in a new membrane (corresponding to level 2), and bind this level-2 membrane to the level-1 membrane (Figure 5a). Note that this specific membrane topology respects the guarantees provided by execution levels: the quota aspect (at level 2) does not see the join points of the browser (at level 0) but only those of the caching aspect. This way the browser can still consume arbitrary disk space.

In essence, execution levels give rise to a restricted topological picture—a linear order between groups of aspects. This makes execution levels not well suited in the general case when there is no such meaningful order; the rest of this section illustrates how membranes go beyond levels.

4.2 Membranes in a tree

Suppose we also want to control the disk consumption of the browser, separately from the disk consumption of the web cache. We can create two instances of the quota aspects, register them in two different membranes, and then bind these membranes to the browser and cache membranes, respectively (Figure 5b). The resulting tree-based composition ensures that one quota aspect observes only the join points of the browser, while the other quota aspect observes only the join points of the cache aspect.

With execution levels, this kind of scenarios cannot be expressed. The fact that the second quota aspect observes the browser means that it resides at level 1; as a consequence, its computation is visible to all aspects deployed at level 2 (the other quota instance). In other words, it is not possible with levels to have an aspect observe only part of the lower level.

4.3 Membranes in a DAG

In the previous scenario, we controlled the consumption of the browser and the web cache separately. If the overall consumption of the complete application is required, a single instance of the quota aspect can be used. This is illustrated in Figure 5c: the membrane in which quota is registered advises *both* the cache and browser membranes.

With execution levels, this scenario means that the quota aspect must observe computation at two levels at the same time. The only way to achieve this is to deploy the same aspect instance at both levels [18]. This however reopens the door to infinite regression, because the aspect deployed at both levels can now observe its own computation. Adopting a graph-based topology allows us to express this scenario without reintroducing conflation.

Supporting a DAG topology is strictly more expressive than the linear topology of levels. In addition, because the graph is acyclic, the properties of execution levels (no conflation, no possibilities for infinite regression due to weaving) are preserved[2].

5. Advanced Topological Scoping

We now show how membranes enable us to express advanced topological scoping features. All these features ensure properties when the topology evolves dynamically. First, in section 5.1 we introduce the notion of co-observation that specifies that different advising membranes receive exactly the same join points. Second, in section 5.2 we explore the use of hierarchical membranes (*i.e.*membranes around membranes) to define "firewalls" for join points. Finally, in section 5.3 we consider the possibility to define crosscutting membranes.

5.1 Co-observation

We introduce a new notion of composition between two advising membranes—coined *co-observation*.

For instance, let us consider a base web browser, a web caching aspect and a logging aspect. These two aspects can

[2] This property vanishes when we introduce crosscutting membranes, as discussed later on.

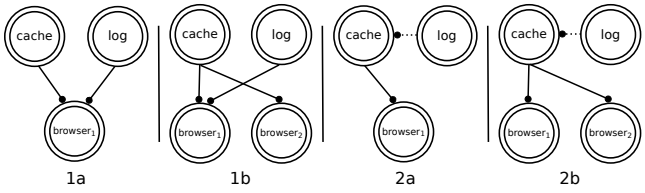

Figure 6. Co-observation in action. 1a) Browser$_1$ is advised by Cache and Log. 1b) Browser$_2$ is dynamically created, it should be advised by both aspects but there is an error. 2a) Browser$_1$ is advised by Cache and Log (as Log co-observes Cache). 2b) Browser$_2$ is dynamically created, it is advised by both aspects (for Log co-observes Cache).

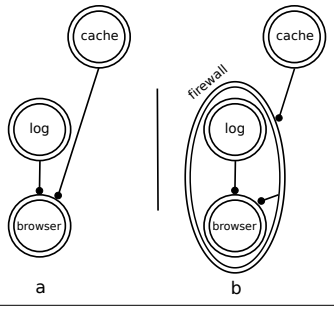

Figure 7. Hierarchical membranes. In b), a firewall membrane prevents secured URLs (HTTPS) from being cached.

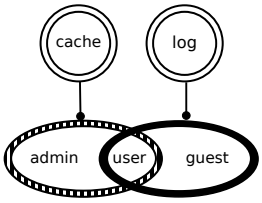

Figure 8. Crosscuting membranes.

be applied to the browser as in scenario 1a of Figure 6 in order to both cache its internet communications and log the URLs it accesses. The cache and log aspects are registered in new membranes: the cache and log membranes, respectively. A membrane is deployed around the browser computation and this membrane is advised by the two aspect membranes. Both aspects have the same pointcut: `get(URL)`. When the web caching aspect lets the base computation actually download a page, the URL is logged for security reason. When a second web browser is dynamically created, we wish the caching aspect *and* the logging aspect are applied to it. However, nothing prevents us from applying only the web caching aspect to the new browser. In scenario 1b, the cache membrane is bound to the membrane of the second browser, but the log membrane is not.

Co-observation offers a more robust solution. In scenario 2a, the membrane of log co-observes (noted with a dotted arrow) the membrane of cache, hence both membranes receive the same join points. This is equivalent to scenario 1a. However, when a new browser and its membrane are dynamically created, the cache membrane only has to be bound to the new browser membrane as in scenario 2b. Indeed, co-observation ensures that advising membranes of both aspects will receive *exactly* the same join points (there is no way to use the cache and not be logged).

5.2 Hierarchical membranes

A membrane can be deployed around any computation. In particular, a membrane can be deployed around a composed system (such as depicted in earlier Figures 5-6). A hierarchical membrane controls propagation of join points produced by the composed system, as well as reception of join points sent by other membranes to the system.

For instance, let us consider in Figure 7a a web browser advised by a log aspect that stores the accessed URLs and a caching aspect that stores a copy of the downloaded pages. For security reason, we can decide secured URLs (*i.e.*HTTPS) should not be cached. This can be performed as depicted in Figure 7b by deploying around the browser and its log aspect a hierarchical membrane (*firewall*) that propagates only join points corresponding to non secured URL accesses. In order to control the propagation of the join points produced by the composed system, the hierarchical membrane has to be bound to the browser and the log membranes (as depicted by the lollypop arrows inside the firewall). The web cache membrane bound to the hierarchical membrane will only receive join points corresponding to non secure requests, hence the cache will only store the non secured pages. Note that, if the membrane of an aspect co-observes the firewall membrane, it will only receive join points that the firewall membrane receives from the outside. So, outside aspects cannot access the secured URLs. Such a membrane plays the role of a firewall for join points. It ensures secured URLs cannot be observed even if new aspects are dynamically introduced.

Also note that when a firewall membrane is introduced, some bindings have to be modified. Bindings cannot cross the firewall membrane. Advising membranes outside of the firewall now have to be bound to the firewall. And the firewall membrane must be bound to its inner sub-membranes. In the current example, the cache membrane is bound to the firewall membrane, and the firewall membrane is bound to the browser and the log membranes.

5.3 Crosscutting membranes

Let us consider two different concerns. We can deploy a membrane around the computations that depend on the first concern, and deploy a second membrane around the computations that depend on the second concern. Such membranes crosscut when some computation happens in both membranes.

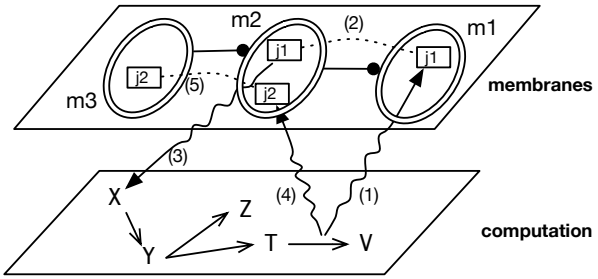

Figure 9. Membranes in action in a level-like setting.

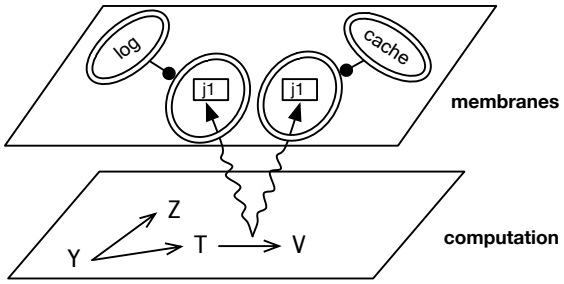

Figure 10. Crosscutting membranes: the same join point appears in two membranes.

For instance, let us consider three base computation: a web browser for an administrator, one for a standard user and one for a guest. For efficiency reasons, the administrator and the standard user browsers must use a web cache. There is no point in caching internet accesses of the guest because we consider they have a short life span. On the other hand, the standard user and guest accesses must be logged for security reasons, while administrator accesses are not. Such a scenario can be expressed with crosscutting membranes, as depicted in Figure 8. A first membrane is deployed around the admin and standard user browser computations, as depicted by the stripped membrane. The cache aspect is registered in another membrane and this cache membrane is bound to the stripped membrane. A second membrane is deployed around the standard user and guest browser computations, as depicted by the black membrane. The log aspect is registered in another membrane and this log membrane is bound to the black membrane. The membranes crosscut for the standard user browser computation, which is concerned with both security and efficiency.

Crosscutting membranes make it simple to maintain the topology of the system when new membranes are created dynamically. For instance, when a new security aspect is created dynamically, it only has registered in a membrane that gets bound to the black membrane. Similarly, ensuring that a new guest browser is executed in the black membrane ensures that all security aspects apply to it.

6. Membranes and Computation

The previous sections have developed various facets of the membrane-based model we propose for flexible aspect scoping. We have seen how programmable membranes themselves can play an important role in controlling how join points get exposed to aspects in a system. We have also shown how the binding relation between membranes enables the definition of flexible topologies. We now zoom in and clarify the relation between membranes and the actual program computation.

6.1 Membranes and computation planes

We have mentioned repeatedly that membranes "contain computation", as opposed to code. This is fundamental. To visualize this fact, we now represent membranes and computation in two separate planes, as in Figure 9. This drawing makes clear that membranes are an overlay on top of computation, solely dedicated to aspect scoping. In the computation plane, arrows and nodes represent a call graph. Squiggling arrows represent the shifts between the computation plane and the membranes plane. Three membranes are depicted, such that m3 is bound to m2, which is bound to m1. This corresponds to the execution levels topology, illustrated previously on Figure 5a.

Figure 9 illustrates the dynamic relation between computation and membranes; this relation is a generalization of the fact that execution levels themselves are a property of the execution flow, not of (static) code artefacts. A given piece of code can produce a join point observed by a membrane, and later a join point observed by a different membrane. Indeed, consider that the program is initiated by deploying m1 over the main expression. When computation flows from T to V, a join point j1 is generated (1), and consequently, j1 appears inside membrane m1. Join point j1 then possibly flows through m1 to inside m2 (2). An aspect registered in m2 is then woven on j1 (3), causing some computation to happen (*e.g.,* its advice computation). If this computation flows again from T to V, another join point j2 is generated (4), but this time it appears inside m2: this reflects the fact that at this point in time, computation "happens inside" m2. Join point j2 is therefore not propagated to m2 for advising, but only to m3 (5). Any aspect registered in m3 is then woven on j2.

6.2 Crosscutting membranes revisited

In Section 5.3 we have presented crosscutting membranes: membranes crosscut each other when some computation is inside each of them. With execution levels [17], such a crosscutting cannot occur (execution is always happening at one level). With membranes, this situation can be permitted, for instance to support the example described in the previous

section, where the user browser computation is included inside a membrane advised by a caching aspect, as well as a membrane advised by a log aspect.

We previously depicted this situation with overlapping membranes. In fact, using a third dimension to separate computation from membranes clarifies what crosscutting membranes really mean (Figure 10): there is no need to depict the membranes as overlapping. For instance, when computation flows from T to V, the *same* join point j1 has to appear inside *both* advised membranes[3]. There are multiple possible semantics for dealing with this joint appearance. A concrete membrane system could choose non-deterministically a unique membrane, or choose the most recent one; it could also compose sequentially (or even concurrently) both membranes, serving as a low-level composition medium.

6.3 Membrane creation and configuration

Our description of programmable membranes for expressive aspect scoping does not commit to any specific API or set of language constructs that deal with membrane creation, deployment and configuration. This is intentional, because the design space in that regard is wide, and it is not our objective to settle for a specific point in that space. It is the responsibility of concrete implementations of membrane-based aspect systems to specify these features precisely. Here, we just briefly describe some possible approaches. Section 7 presents a particular implementation for Scheme.

Membrane creation and configuration could all be done statically, similarly to the extension of AspectJ with execution levels proposed recently [18]. Membranes are defined statically by specifying the aspects that are registered in each of them, and by describing how membranes are bound together. An initial startup membrane is created for the main program. With such a static approach, the same efficient implementation technique as that used for execution levels can be applied.In addition, it can be possible to ensure that membranes never crosscut each other.

It is also possible to support a much more dynamic set of mechanisms for membrane creation and deployment, such as creating a new membrane object and explicitly spawning some computation in it, similarly to how dynamically-scoped aspects are deployed in many aspect languages; or deploying membranes explicitly on objects. It could even be possible to define membrane deployment more intentionally, by specifying activation predicates (*e.g.,* all computation that is in the control flow of X and not Y is considered to be inside m). Of course, dynamic creation and deployment of membranes can easily produce crosscutting membranes. Yet another possibility is to design a DSL for specifying possibly dynamic topologies, and support static analyses to enforce properties of the topology, such as acyclicity or the absence of problematic crosscutting membranes.

[3] As the same join point can now appear inside two membranes, the properties of execution levels (no conflation, no possibilities for infinite regression due to weaving) are no longer preserved, even for acyclic topologies. With this multiple deployment—which corresponds to an aspect executing at several levels in execution levels—an advising membrane that is crosscutting with one of its advised membranes is exposed to its own join points.

7. A Scheme Implementation of Membranes for AOP

To illustrate a particular point in the design space of membrane-based aspect languages, we have developed MAScheme—an extension of AspectScheme [7] with membranes. In the current version, membranes in MAScheme support topological scoping as well as a limited form a programmability, namely to control the visibility of their computation. Section 7.1 shows how membranes are created and deployed in MAScheme, and how aspects are registered with membranes. Section 7.2 shows how the different topological scenarios of Section 4 are expressed in MAScheme. Finally, Section 7.3 explains how membranes with a programmable visibility filter make it possible to enforce encapsulation *a la* Open Modules [1].

Due to space limitation, we do not expose the implementation details of MAScheme here. The MAScheme implementation and examples of this section are available online: http://pleiad.cl/research/mascheme

7.1 Creating and deploying membranes

MAScheme includes primitives for creating, deploying and binding membranes, as well as for registering aspects with membranes.

Membranes are created with `(new-membrane)`:

```
(define m0 (new-membrane))
(define m1 (new-membrane))
```

Aspects in MAScheme are specified with a pointcut and an advice function, like in AspectScheme. The difference is that aspects now have to be registered in membranes:

```
(register (call get-url) cache m1)
```

This expression registers a `cache` advice in membrane `m1`, with a pointcut that matches calls to the `get-url` function.

The following deploys membrane `m0` around a browser execution:

```
(deploy-fluid m0 (browser))
```

`deploy-fluid` specifies that all the join points occurring within the dynamic extent of the execution of `browser` happen "inside" membrane `m0`. Of course, because membrane `m0` is not advised by any membrane so far, no aspect apply. If prior to the above, `m1` (in which the caching aspect is registered) is made to advise `m0`:

```
(advise m1 m0)
```

then the execution of the browser is cached.

Deploying a membrane with `deploy-fluid` gives it dynamic scoping. Similarly to the aspect deployment features of AspectScheme [7], membranes in MAScheme can also be deployed with lexical scoping. For instance, we could define `browser` as follows:

```
(define browser
  (deploy-static m0
    (lambda ()
      (get-url "http://foo.com"))))
```

Membrane `m0` now wraps all computation that is lexically in the body of `deploy-static`. This means that `m0` is "captured" in the lambda, and sees the call to `get-url` each time `browser` is applied. It is possible to go beyond `deploy-static` and `deploy-fluid` using scoping strategies [15, 16].

Finally, MAScheme supports crosscutting membranes, since different membranes can be deployed around the same computation. Crosscutting membranes are composed sequentially exactly like multiple aspects are composed in AspectScheme, *i.e.* in reverse order of deployment [7].

7.2 Topological scoping in MAScheme

We now show how the basic topological scoping examples of Section 4 are expressed in MAScheme[4]. In these scenarios, `browser` is a function that in turn calls `get-url` and `access-disk`. `cache` and `quota` are advices for the caching and quota aspects, respectively.

Tower. We first create the three membranes corresponding to the three execution levels, then bind advising membranes following the tower topology:

```
(let ((l0 (new-membrane))  ;; level-0
      (l1 (new-membrane))  ;; level-1
      (l2 (new-membrane))) ;; level-2
  (advise l1 l0)
  (advise l2 l1)
```

Then, the caching aspect is registered with level 1, and quota with level 2.

```
(register (call get-url) cache l1)
(register (call access-disk) quota l2)
```

Finally, the browser is run at level 0:

```
(deploy-fluid l0 (browser))
```

Disk accesses performed by the caching aspects are advised by the quota aspect, but the quota aspect does not see disk accesses performed by the browser.

Tree. In this scenario, a fourth membrane `l3` is created and made to advise the browser computation, and the quota aspect is also registered with it:

```
...
(advise l3 l0)
(register (call access-disk) quota l3)
...
```

Now the disk consumption of the browser is monitored, separately from that of the web cache.

[4] The MAScheme examples are fully presented in Appendix A and are available online (also including an illustration of crosscutting membranes, following Figure 8).

```
structure Math = struct
  val fib = fn x:int => 1;

  around call(fib) (x:int) =
    if (x > 2)
    then fib(x-1) + fib(x-2)
    else proceed x;

  (* advice to cache calls to fib *)
  val inCache = fn ...;
  val lookupCache = fn ...;
  val updateCache = fn ...;

  pointcut cacheFunction = call(fib);
  around cacheFunction(x:int) =
    if (inCache x)
    then lookupCache x
    else let v = proceed x
         in updateCache x v; v
end :> sig
    fib : int->int
```

Figure 11. Fibonacci caching example with Open Modules (adapted from [1]).

DAG. The DAG scenario is similarly simple to obtain. It suffices to change the advising declarations above to:

```
...
(advise l1 l0)
(advise l2 l0)
(advise l2 l1)
...
```

This way, `l2` advises both `l0` and `l1`, ensuring that the quota aspect controls both the browser computation and the caching aspect.

7.3 Encoding Open Modules with programmable membranes

Open Modules [1] have been proposed to consider the base computation as a grey box rather than a glass box. This is achieved by considering the base computation inside (ML-like) modules that can choose to expose only some join points to the outside. This encapsulation enables to reason formally and modularly about observational equivalence of programs under advice. In this section, we sketch how to encode open modules using programmable membranes in MAScheme with the main example used by Aldrich [1].

The Fibonacci caching example, described in Figure 11, is a module that computes the Fibonacci function. The function `fib` is implemented as its base case, returning 1. Then, an around advice intercepts any calls to `fib` to handle the recursive part. The rest of the module describes an aspect that caches calls to `fib`, thus allowing the exponential function to run linearly (the cache data structure is not described).

The `fib` function is then encapsulated inside the `Math` module. The last line of the definition of this module describes which join points are exposed—in this case, only calls to `fib` can be intercepted from the outside. This means

that a computation outside the `Math` module is not able to tell whether caching has been applied or not. Thus, from an observational equivalence point of view, this implementation of the Fibonacci function inside the module `Math` is equivalent to a purely functional implementation.

MAScheme makes it possible, when creating a membrane, to specify a *filter* function from join points to booleans which tells when a joint point is exposed to advising membranes. This mechanism can be used to achieve a similar purpose as Open Modules. To illustrate the encoding of Open Modules in MAScheme, we introduce a simple representation of a module as a pair of a function, and a membrane:

```
;; simple representation of a module
;; = function + membrane
(define-struct module (fun membrane))
```

We now define the `Math` "module". As in [1], the definition of the `fib` function in itself is simply given by a constant function that is advised by two aspects, here called `fib_rec` and `cache` (provided in Appendix A). The encoding then consists of specifying two crosscutting membranes. First, an internal membrane `int_memb`, transparent, allows internal aspects (here, `fib_rec` and `cache`) to advise any join point inside the module. Second, an external membrane `ext_memb`, translucent, propagates only join points described in the signature of the module, as specified by the filter function `(call fib)` (a pointcut):

```
;; the Math module
(define Math
  (let* ((fib (lambda (n) 1))
         (int_memb (new-membrane))
         (ext_memb (new-membrane (call fib))))
    (advise int_memb int_memb)
    (register (call fib) fib_rec int_memb)
    (register (call fib) cache int_memb)
    (module
        (lambda (n)
          (deploy-fluid ext_memb
            (deploy-fluid int_memb
              (fib n))))
        ext_memb)))
```

The external membrane `ext_memb` is designed to expose only calls to `fib`. Join points occurring inside `ext_memb` but not matching `call(fib)` are invisible to advising membranes.

The module `Math` consists of the `fib` function together with the `ext_memb` membrane. Therefore, a client can import (deconstruct) the module `Math`, advise its external membrane, and use its function:

```
;; importing the module Math
(let ((fib      (module-fun Math))
      (fib_memb (module-membrane Math)))
  ;; advising it
  (let ((log_memb (new-membrane)))
    (register call? logging log_memb)
    (advise log_memb fib_membrane)
    ;; using fib
    (fib 10)))
```

The `logging` aspect registered in the `log_memb` membrane cannot observe the internal computation of the `Math` module: even though its pointcut `call?` matches *any* call join point, it only sees one. Of course, this exercise is just a proof-of-concept. Properly integrating membranes with a real module system, like that of Racket (the dialect of Scheme in which MAScheme is implemented), is a topic of future work.

8. A Kell Implementation of Membranes for AOP

While MAScheme is a good prototype to explore the power of topological scoping and translucent membranes, it does not fully support programmable membranes and their applications sketched in Section 3. In this section, we leave aside the sequential world and explore an instantiation of programmable membranes for AOP in the Kell calculus of Schmitt and Stefani [12].

The motivation for doing so is multifold. First, because our inspiration comes in part from work on membrane computing and calculi, it is natural to see how using a language with an explicit notion of membrane eases the task. In that respect, the Kell calculus has the proper level of abstraction; in contrast, the generic membrane model of Boudol [5] is too abstract for our purpose here.

Second, the Kell calculus directly supports programmable membranes in a naturally concurrent and distributed setting. It is built around a π-calculus core, and follows five design principles, which are essential in distributed and mobile programming: hierarchical localities, local actions, higher-order communication, programmable membranes, and dynamic binding. Therefore, using the Kell calculus naturally provides the potential to deal with taming aspects in presence of concurrency and distributed computing and reasoning principles based on bisimulations; although we have not yet explored these tracks (see Section 9.2 for a brief discussion).

Third, the Kell calculus is not only a foundational model for distributing programming; it has been concretely implemented [3]. It can thus be a solid base for implementing our framework on top of existing AOP systems.

For the reader not familiar with the Kell calculus, Section 8.1 gives a brief introduction, including the basic formulation of programmable membranes. While it already has programmable membranes, the Kell has no notion of aspect weaving; our contribution in this section is then to formulate a precise protocol between membranes and computation (Section 8.2), and to define in the Kell calculus several membranes discussed previously (Section 8.3).

8.1 The Kell calculus in a nutshell

Most of this subsection is only a condensed (slightly adapted) version of the reference description of the Kell cal-

culus [12], included for convenience. We refer the reader to [12] for a more detailed description of the Kell calculus.

Syntax of processes. The Kell calculus—whose central notion is that of processes communicating on ports—has the same basic constructs as the π-calculus, which is a reference for concurrent computing [9]. In particular, there are:

- *Output on port:* $a\langle v\rangle.T$ is an output on port a with argument v and continuation process T.
- *Input on port:* $\xi \triangleright P$ is an input on port a, where ξ is an input pattern (*e.g.,* $a\langle x\rangle$ with x an input variable), and P is a process. An input pattern can possess an annotation $^{\uparrow}$ (resp. $^{\downarrow}$) to indicate that the message will be received from the outside (resp. inside) of the kell.
- *Permanent input on port:* $\xi \diamond T$ is a permanent input[5] on port a, where ξ is an input pattern and P is a process.
- *Parallel composition:* $(P \mid Q)$ denotes the parallel composition of processes P and Q.
- *Restriction:* $\nu a.P$ denotes the creation of a fresh name a whose initial scope is P.
- *Null process:* 0 is the process that does not perform any action.

The main reduction rule is the message reception

$$a\langle Q\rangle.T \mid (a\langle x\rangle \triangleright P) \rightarrow T \mid P\{Q/x\}$$

which in case of $\triangleright$, consumes the input, while for $\diamond$, the input remains in the solution.

Hierarchical localities. Name of localities in the Kell calculus are called *kells*. A locality, noted $k[P]$, means that the process P executes at kell (*a.k.a.*location) k. The structure of kells is hierarchical. In the basic setting of the Kell calculus, a message $a\langle x\rangle$ can go across a kell (for example moving up) only if the environment of the kell possesses the trigger

$$a\langle x\rangle^{\downarrow} \triangleright P.$$

As it is not suitable in our setting, we introduce syntactic sugar (only when executing inside a programmable membrane)

$$\xi \diamond P^{\uparrow} \qquad \text{and} \qquad \xi \diamond P^{\downarrow}$$

meaning that the membrane is emitting the process P upwardly or downwardly.

Programmable membranes. In the Kell calculus, the computation of a process P is located in a kell. All non local communications must explicitly go through each kell and Schmitt and Stefani exploit this fact to introduce an intermediate kell, whose role is to control communications, the whole thing forming a membrane. A programmable membrane

$$\nu l.l[M \mid k[P]]$$

[5] Permanent inputs can be encoded in a more basic version of the Kell calculus, but we introduce them explicitly to ease membranes definitions.

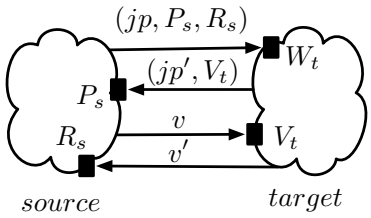

Figure 12. Weaving protocol, with communication ports.

on the kell $k[P]$ is thus encoded as a new external kell l together with a communication process M that drives communication to and from the kell k.

8.2 Protocol between membranes and computations

We now introduce the protocol used between computations and membranes. We use here a particular protocol based on the notion of around advice, as introduced in Figure 1: aspects can do some computation, then proceed (zero, one, or more times), and then do some final computation. Of course, this is not the only way aspect and base computation can interact with each other and our framework gives the freedom to implement any protocol of interest. For instance, in a strongly concurrent setting, it would make sense to define a protocol where asynchronous advice is supported.

Weaving protocol. The weaving protocol is described in Figure 12, which refines Figure 1 with communication ports from the Kell calculus. Ports are represented as black rectangles. We call $source$ the computation that is generating a join point and $target$ the advising computation.

- When computation reaches a point of interest, a join point `jp` is emitted on communication port `weave` (W_t), together with two fresh callback communication ports `proceed` and `resume` (P_s and R_s).
- When $source$ receives on its `proceed` port (P_s) a join point `jp`$'$ and a port `value` (V_t), it executes the original computation (with the parameters contained in `jp`$'$), then sends on `value` the value `v` computed by the original computation.
- Finally, when $source$ receives on `resume` a value `v`$'$, it resumes its execution after the current point of interest, using `v`$'$.

These events occur in a particular order. The protocol of the base interface can be described by the following regular expression:

$$\texttt{weave} < \texttt{jp}, \texttt{proceed}, \texttt{resume} >^{\uparrow}$$
$$(\texttt{proceed} < \texttt{jp}', \texttt{value} >^{\downarrow} \texttt{value} < \texttt{v} >^{\uparrow})^{*}$$
$$\texttt{resume} < \texttt{v}' >^{\downarrow}$$

In the style of the Kell calculus, arrows in the expression above indicate when the computation receives (down arrow) or emits (up arrow) a message.

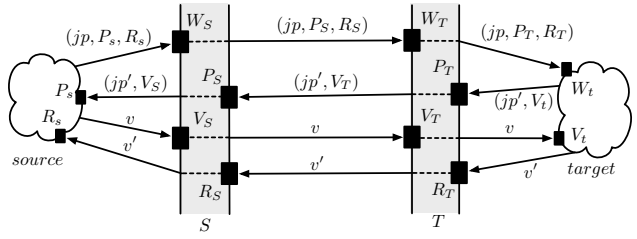

Figure 13. Weaving protocol with two membranes.

Note that the original computation can be executed zero or many times. Also, only the `weave` port has to be known upfront to initiate the communication; the `proceed`, `resume`, and `value` ports are freshly created for each instantiation of the protocol.

Dually, each advising computation implements the interface:

- An advice starts its execution when it receives on `weave` a join point `jp` and two callback ports `proceed` and `resume`.
- It stops before the proceed instruction and sends on `proceed` the join point $\texttt{jp}'$ (that can contain new modified arguments) and the port `value`.
- Its execution is resumed after the proceed instruction by receiving a value `v` on `value`.
- Finally, at the end of the advice, it sends a value $\texttt{v}'$ on `resume`.

The protocol of the advising interface can be described by the following regular expression; which is obtain from the previous one by simply flipping arrows:

$$\begin{array}{l}\texttt{weave} < \texttt{jp}, \texttt{proceed}, \texttt{resume} >^{\downarrow} \\ (\texttt{proceed} < \texttt{jp}', \texttt{value} >^{\uparrow} \texttt{value} < \texttt{v} >^{\downarrow})^* \\ \texttt{resume} < \texttt{v}' >^{\uparrow}\end{array}$$

Membranes. Membranes are introduced as extra levels of indirection in the weaving protocol. Wrapping the base computation *source* and the advising computation *target* into membranes allows control over propagation of and effects on join points. Figure 13 depicts the weaving protocol with two membranes, S deployed around *source*, and T deployed around *target*. Note that from the point of view of *source* and *target*, the protocol is unchanged: S acts exactly as *target* from the point of view of *source*, and vice-versa.

Taken individually, each membrane combines both protocols. As an advised membrane, S plays the role of an advising computation for its inside (*source*), and plays the role of an advised computation for its outside (T). Dually, as an advising membrane, T plays the role of an advising computation for its outside (S), and plays the role of a base computation for its inside (*target*).

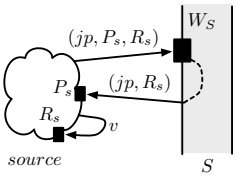

Figure 14. An opaque advised membrane.

8.3 Defining membranes in the Kell calculus

We now describe the definition of different kinds of membranes in the Kell calculus. We only present how to define advised membranes; advising membranes are defined dually by reverting the sense of emission. In the sequel, every membrane has a unique permanent port `weave`, which interprets messages from the inside as join point emission, and messages from the outside as join point reception.

Transparent membrane. The transparent advised membrane $\texttt{M}_{\texttt{transparent}}(\texttt{weave}_{\texttt{in}}, \texttt{weave}_{\texttt{out}})$—which receives join points on $\texttt{weave}_{\texttt{in}}$ and communicates them on port $\texttt{weave}_{\texttt{out}}$ of a given advising membrane—is defined by the following Kell program:

$$\begin{array}{l}\texttt{M}_{\texttt{transparent}}(\texttt{weave}_{\texttt{in}}, \texttt{weave}_{\texttt{out}}) = \\ \quad \texttt{weave}_{\texttt{in}}\langle \texttt{jp}, \texttt{proceed}, \texttt{resume}\rangle^{\downarrow} \diamond \\ \qquad \texttt{weave}_{\texttt{out}}\langle \texttt{jp}, \texttt{proceed}, \texttt{resume}\rangle^{\uparrow} \\ \qquad \mid \texttt{proceed}\langle \texttt{jp}', \texttt{value}\rangle^{\uparrow} \diamond \texttt{proceed}\langle \texttt{jp}', \texttt{value}\rangle^{\downarrow} \\ \qquad\qquad\qquad\qquad \mid \texttt{value}\langle \texttt{v}\rangle^{\downarrow} \triangleright \texttt{value}\langle \texttt{v}\rangle^{\uparrow} \\ \qquad \mid \texttt{resume}\langle \texttt{v}\rangle^{\uparrow} \triangleright \texttt{resume}\langle \texttt{v}\rangle^{\downarrow}\end{array}$$

The membrane $\texttt{M}_{\texttt{transparent}}(\texttt{weave}_{\texttt{in}}, \texttt{weave}_{\texttt{out}})$ receives messages on port $\texttt{weave}_{\texttt{in}}$ initiated by a sub-membrane and then (in parallel):

- emits $\texttt{weave}_{\texttt{out}}\langle \texttt{jp}, \texttt{proceed}, \texttt{resume}\rangle$ upward, a message on port $\texttt{weave}_{\texttt{out}}$ to propagate the join point `jp` and fresh callback ports `proceed` and `resume` to a given advising membrane
- listens on port `proceed` a call to proceed from the advise. In that case, it emits downward the call to proceed to the base computation and triggers from port `value` the result from the inside to the outside of the membrane.
- creates a trigger for port `resume` from the outside to the inside of the membrane

The astute reader will have noticed that the interaction described here uses less ports than depicted in Figure 13. For instance, ports P_T and P_S are simply aliases to P_s (which corresponds to `proceed`). This is safe, because of explicit migration: the membrane still acts as an intermediary and cannot be bypassed.

We now turn to the definition of less transparent membranes as introduced in Section 3.

Opaque membrane. Opaque membranes never let a join point traverse through them. Therefore, an opaque advised membrane protects the computation inside it from aspect advising. Such an advised membrane can simply be defined by

directly sending a call to proceed to the computation, with the continuation port `resume` itself. Then the computation proceeds and publishes the return value on its own `resume` port, which makes it go through. An opaque advising membrane is illustrated on Figure 14. Its definition is:

$$\mathtt{M}_{\mathtt{opaque}}(\mathtt{weave}_{\mathtt{in}}) = \mathtt{weave}_{\mathtt{in}}\langle \mathtt{jp}, \mathtt{proceed}, \mathtt{resume}\rangle^{\downarrow} \diamond \mathtt{proceed}\langle \mathtt{jp}, \mathtt{resume}\rangle^{\downarrow}$$

Translucent membrane. Translucent membranes are just a mix between opaque and transparent membranes parametrized by a join point filter. For each join point received on port $\mathtt{weave}_{\mathtt{in}}$, a translucent advised membrane decides to return directly to the computation or to make the join point (or an altered version of the join point) visible to advising membranes.

Immutable membrane. Immutable advising membranes are transparent membranes augmented with safety policies. Namely, an immutable advising membrane enforces exactly one call to proceed, with a fixed join point, and a fixed return value. This is done by making the computation in the membrane itself more sequential, in particular by waiting for a call to proceed before waiting for a call to resume.

$$\begin{aligned}&\mathtt{M}_{\mathtt{immutable}}(\mathtt{weave}_{\mathtt{in}}, \mathtt{weave}_{\mathtt{out}}) = \\ &\quad \mathtt{weave}_{\mathtt{in}}\langle \mathtt{jp}, \mathtt{proceed}, \mathtt{resume}\rangle^{\downarrow} \diamond \\ &\qquad \mathtt{weave}_{out}\langle \mathtt{jp}, \mathtt{proceed}, \mathtt{resume}\rangle^{\uparrow} \mid \\ &\qquad \mathtt{proceed}\langle \mathtt{jp}', \mathtt{value}\rangle^{\uparrow} \triangleright \\ &\qquad\quad \mathtt{proceed}\langle \mathtt{jp}, \mathtt{value}\rangle^{\downarrow} \mid \\ &\qquad\quad \mathtt{value}\langle \mathtt{v}\rangle^{\downarrow} \triangleright \mathtt{value}\langle \mathtt{v}\rangle^{\uparrow} \mid \\ &\qquad\qquad \mathtt{resume}\langle \mathtt{v}'\rangle^{\uparrow} \triangleright \mathtt{resume}\langle \mathtt{v}\rangle^{\downarrow}\end{aligned}$$

The call to proceed transmitted inside the membrane is done with the original join point `jp` instead of the potentially new join point $\mathtt{jp}'$ received from the advice. Furthermore, the fact that the triggering rule for `proceed` is linear (using $\triangleright$ instead of $\diamond$) forces exactly one call to proceed. Finally, the value `v` returned by the original computation on port `value` is conserved and used to resume the original computation—instead of the potentially new value $\mathtt{v}'$.

9. Putting membranes in perspective

9.1 Modularity and encapsulation

Programmable membranes can express most interesting proposals for protecting base code from unwanted advising. Open Modules [1] make it possible to define that only certain public pointcuts are advisable by aspects outside the module. As illustrated in Section 7.3, translucent membranes can encode open modules by exposing only the join points corresponding to these public pointcuts. In the same way as Aldrich uses logical equivalence to justify modular reasoning, we can use bisimulation techniques at the level of membranes. A fundamental difference between open modules and membranes is that membranes can be dynamic. EffectiveAdvice [10] uses monads to explicitly reason about the effects of advice. Membranes can enforce restrictions on computational effects like number of calls to proceed, as well as modification of arguments and return values, etc. Interestingly, membranes support these restrictions even in presence of quantification (pointcuts), which is missing in EffectiveAdvice. On the other hand, because EffectiveAdvice uses monads, it can statically enforce these restrictions, and also supports arbitrary effects. Extending membranes in these directions is an interesting perspective.

Beyond existing proposals, membranes suggest new mechanisms. Most interestingly, membranes can not only be used to protect the advised computation, but also the advising one. In other words, it is possible to protect aspects from seeing unwanted join points. This can be particularly useful in two scenarios. First, we have seen that in presence of crosscutting membranes, infinite regression can happen; an advising membrane can filter out reentrant join points to avoid this. Generalizing, an advising membrane can also filter out join points for security reasons, for instance join points produced by untrusted threads, or under suspicious conditions. To the best of our knowledge, this dual view on encapsulation has not been explored elsewhere.

Because membranes control both emission and reception of join points, they can as well be used to raise the level of abstraction of join points to domain-specific join points, similarly to the mechanisms for explicit custom event announcements in Ptolemy and IIIA [11, 14]. The advantage of using membranes is that custom join points can be generated by the membranes themselves (either upon reception or upon emission), thereby preserving the implicit nature of join points. This suggests that membranes can combine aspect-based and event-based systems in a unified framework with flexible topologies.

9.2 Towards concurrent and distributed membranes

As mentioned at the beginning of Section 8, we believe that membranes constitute a convenient framework for AOP systems integration in a concurrent and distributed settings. Indeed, the fact that several planes of computation can be integrated through the membrane plane makes it possible to advise distinct threads or remote machines without changing the computational planes. During the weaving, each order to proceed, return a value or resume is done on a fresh private port between the advised membrane and its advising membrane. Thus, executions can be completely concurrent while preventing from name mismatch during the weaving process. Of course, synchronization on shared states remains necessary (*e.g.,* when a stateful aspects can advise several computation at the same time).

"Pointcutlets". Taking distribution into account, a membrane can be asked to pre-compute some (logical) part of a pointcut on behalf of a remote aspect. Such a "pointcutlet" makes it possible to reduce communication of join points along the network, limiting the set of transmitted join points

to those that have a chance to be matched by the remote aspect. A pointcutlet can be loaded by the advised membrane when an advising membrane asks for binding. The pointcutlet can also evolve in time. For instance, an advice can send (using higher-order communication features of the language) some updates to the pointcutlet in order to filter the next join point. This can be particularly useful for stateful aspects that match a sequence of join points. Indeed, the pointcuts of stateful aspects change with their state. A particularly interesting case is that of a remote trace matching aspect, which ships (part of) its state machine to the advised membrane, so that the state machine can advance locally.

10. Conclusion

This paper proposes to adapt the notion of programmable membranes [5, 12] in order to control the controversial power of aspects. We describe a notion of membranes in an AOP context, where membranes are just an overlay on top of computation, dedicated to controlling propagation of join points and local aspect weaving. In addition, the topological relation between advising and advised membranes supports flexible scoping scenarios, subsuming execution levels [17]. The notion of membranes for AOP proposed in this paper can be instantiated in a variety of ways. We have detailed two initial technical treatments of membranes for aspects. First, to illustrate membrane deployment, configuration and topological scoping, we have developed an extension of AspectScheme with membranes, MAScheme. MAScheme also supports a form of translucent membranes, used to encode Open Modules. Second, in the frame of the Kell calculus, we have illustrated a particular weaving protocol and membrane programming techniques. With respect to the development of MAScheme, a particular area of future work is the integration with the module system of Racket. We are also looking at instantiating the Kell-based approach for specific aspect languages. More generally, while it is clear that this paper leaves many questions unresolved, we believe that unleashing the potential of membranes for modular reasoning, encapsulation, security, and scoping of aspects, including in presence of concurrency and distribution, opens many venues for future investigation.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. Black, editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, number 3586 in Lecture Notes in Computer Science, pages 144–168, Glasgow, UK, July 2005. Springer-Verlag.

[2] *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[3] P. Bidinger, A. Schmitt, and J. Stefani. An abstract machine for the kell calculus. *Formal Methods for Open Object-Based Distributed Systems*, pages 31–46, 2005.

[4] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, July 20-24 2008*, pages 155–165, New York, NY, USA, 07 2008. ACM.

[5] G. Boudol. A generic membrane model (note). In *Global Computing Workshop*, volume 3267 of *Lecture Notes in Computer Science*, pages 208–222. Springer-Verlag, 2005.

[6] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 1996.

[7] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

[8] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[9] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge Univ Pr, 1999.

[10] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [2], pages 109–120.

[11] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 155–179, Paphos, Cyprus, july 2008. Springer-Verlag.

[12] A. Schmitt and J. Stefani. The Kell calculus: A family of higher-order distributed process calculi. In *Global Computing*, pages 146–178. Springer, 2005.

[13] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, Jan. 1984.

[14] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, 20(1):Article 1, June 2010.

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

[16] É. Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009)*, pages 3–14, Orlando, FL, USA, Oct. 2009. ACM Press.

[17] É. Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [2], pages 37–48.

[18] É. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 113–122, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

A. MAScheme Examples

```
#lang racket
(require "mascheme.rkt")



;; Basic utility functions
(define (browser)
  (get-url "http://test.url")
  (access-disk))

(define (get-url url)
  (write 'Url))

(define (access-disk)
  (write 'Disk))

(define (check-cache)
  (access-disk))



;; caching aspect
(define (((cache proceed) . ctx) . args)
  (write 'Cache)
  (check-cache)
  (apply proceed args))



;; quota aspect
(define (((quota proceed) . ctx) . args)
  (write 'Quota)
  (apply proceed args))
```

```
;; Figure 5a: tower
(let ((l0 (new-membrane))
      (l1 (new-membrane))
      (l2 (new-membrane)))
  (advise l1 l0)
  (advise l2 l1)
  (register (call get-url) cache l1)
  (register (call access-disk) quota l2)
  (deploy-fluid l0 (browser)))

;; Figure 5b: tree
(let ((l0 (new-membrane))
      (l1 (new-membrane))
      (l2 (new-membrane))
      (l3 (new-membrane)))
  (advise l1 l0)
  (advise l2 l1)
  (advise l3 l0)
  (register (call get-url) cache l1)
  (register (call access-disk) quota l2)
  (register (call access-disk) quota l3)
  (deploy-fluid l0 (browser)))

;; Figure 5c: graph
(let ((l0 (new-membrane))
      (l1 (new-membrane))
      (l2 (new-membrane)))
  (advise l1 l0)
  (advise l2 l0)
  (advise l2 l1)
  (register (call get-url) cache l1)
  (register (call access-disk) quota l2)
  (deploy-fluid l0 (browser)))
```

Figure 15. Topological scoping examples in MAScheme

```
#lang racket
(require "mascheme.rkt")


;; simple representation of a module
;; = function + membrane
(define-struct openModule (fun membrane))

;; cache data structure
(define cacheStruct (make-hash))

(define (inCache key hash)
  (hash-has-key? hash key))

(define (lookupCache key hash)
  (hash-ref hash key))

(define (updateCache key value hash)
  (hash-set! hash key value))

;; logging aspect
(define (((logging proceed) . ctx) . args)
 (let ((value (apply proceed args)))
   (printf "~a -> ~a is logged~n" args value)))

;; importing the module Math
(let ((fib      (openModule-fun Math))
      (fib_memb (openModule-membrane Math)))
  ;; advising it
  (let ((log_memb (new-membrane)))
    (register call? logging log_memb)
    (advise log_memb fib_memb)
    ;; using fib
    (fib 10)))
```

```
;; the Math module
(define Math

  (let* ((fib (lambda (n) 1))
         (int_memb (new-membrane))
         (ext_memb (new-membrane (call fib))))


    ;; Fibonacci recursion aspect
    (define (((fib_rec proceed) . ctx) . args)
      (let ((arg (first args)))
        (if (> arg 2)
            (+ (fib (- arg 1)) (fib (- arg 2)))
            (apply proceed args))))


    ;; caching aspect
    (define (((cache proceed) . ctx) . args)
      (let ((arg (first args)))
        (if (inCache arg cacheStruct)
            (lookupCache arg cacheStruct)
            (let ((value (apply proceed args)))
              (begin (updateCache arg value cacheStruct))
              value))))

    (fib_rec (lambda arg (fib_rec arg)))
    (advise int_memb int_memb)
    (register (call fib) fib_rec int_memb)
    (register (call fib) cache int_memb)
    (openModule
     (lambda (n) (deploy-fluid int_memb
                   (deploy-fluid ext_memb
                                 (fib n))))
     ext_memb)))
```

Figure 16. Fibonacci caching example (with Open Modules) in MAScheme