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 leverages the concept of *programmable membranes* of Boudol, Schmitt and Stefani, as a means to tame aspects by customizing the semantics of aspect weaving locally. Membranes have the potential to subsume previous proposals in a uniform framework. Because membranes give structure to computation, they enable flexible scoping of aspects; because they are programmable, they enable visibility and safety constraints, both for the advised program and for the aspects. The power and simplicity of membranes open interesting perspectives to unify multiple approaches that tackle the unrestricted power of aspects.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages, Design

Keywords: Aspect-oriented programming, 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. Some proposals make it possible to control the *scope* of aspects; the scope of an aspect is defined as the set of join points against which its pointcuts are matched. Examples include statically and dynamically-scoped aspects [4], scoping strategies [12], as well as execution levels [13]. These proposals can be seen as ways to reduce the impact of an aspect on the aspect deployment side. 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 [6], IIIA [11], and Join Point Interfaces [7]. Yet other proposals rely on types to control the *effects* that aspects can induce [8], or on behavioral contracts [2].

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 [3] and Schmitt and Stefani [10], 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.

Consider 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.

The basic idea is to wrap the program and the aspect 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

FOAL'12, March 26, 2012, Potsdam, Germany.

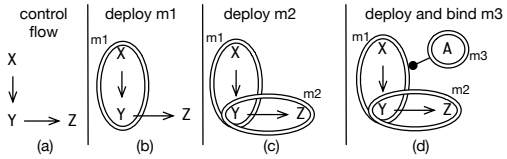

Figure 1. Membranes deployment and aspectual bindings. Membrane m1 (resp. m2) wraps computation X and Y (resp. Y and Z). Membrane m3 is bound to m1: join points from m1's computation will be visible to m3's aspect, A.

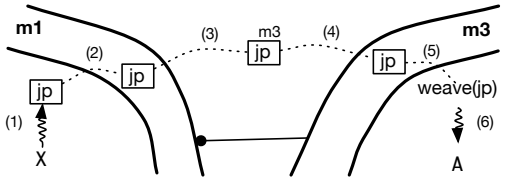

Figure 2. From join point emission to aspect weaving. Computation X produces a join point (1), which travels through membranes m1 and m3 (2-5) before being processed by aspect A (6).

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 [13] 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 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. A companion report [15] contributes both a Scheme implementation and a formal definition of membranes in the Kell calculus [10].

2. Programmable Membranes

We propose to adapt the notion of programmable membranes [3, 10] 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.

2.1 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. 1(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 3).

2.2 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 2 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 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).

2.3 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 3. 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 allows for opacity to be dynamic, based on characteristics of the execution environment.

Programmable membranes can also be used to enforce safety policies that limit the power of aspects, as in [8], 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 2). 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.

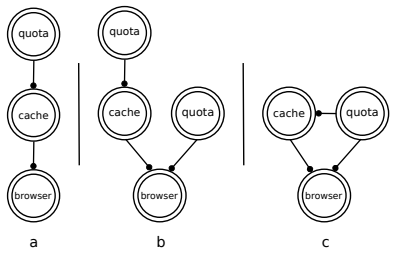

Figure 3. Topological scoping. (a) tower—corresponds to execution levels; (b) tree; (c) DAG.

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

3. Topological Scoping

In addition to enabling safety properties, membranes make it possible to express flexible *topologies* of computation, going beyond execution levels [13]. This section shows how basic and more advanced topological scoping can be achieved.

3.1 Basic topological scoping

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. Suppose now 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 3a). 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, so the browser can consume arbitrary disk space.

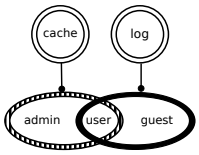

Figure 4. Crosscuting membranes.

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.

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 3b). 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.

Membranes in a DAG. 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 3c: 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 [14]. 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 with no cycle allows us to express this scenario without reintroducing conflation.

3.2 Crosscutting membranes

It is also possible to devise more advanced topological mechanisms for membranes. We now discuss *crosscutting* membranes. In [15] we also describe hierarchical membranes and a relation of co-observation between membranes.

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. Such membranes crosscut when some computation happens in both 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. If we allow membranes to crosscut each other, such a scenario can be directly expressed (Figure 4). A first membrane is deployed around the admin and standard user browser computations. The cache aspect is registered in another membrane and this cache membrane is bound to the first membrane. A second membrane is deployed around the standard user and guest browser computations. The log aspect is registered in another membrane and this log membrane is bound to the second membrane. The membranes crosscut for the standard user browser computation, which is concerned with both security and efficiency.

The flexibility provided by crosscutting membranes is a double-edged sword. Because of their overlap, crosscutting membranes reintroduce the possibility to run into infinite loops: an advising membrane that is crosscutting with one of its advised membranes is exposed to its own join points.

4. Language Support for Membranes

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.

Membrane creation and configuration could all be done statically, similarly to the extension of AspectJ with execution levels [14]. 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.

To date, we know of two aspect languages so far that support membranes, both in the dynamic end of the spec-

trum. MAScheme [15] is our implementation of membranes and aspects for Scheme, which supports dynamic deployment of membranes and different scoping semantics. PHANtom [5] is a new aspect extension of Pharo Smalltalk that integrates membranes. It supports both per-object membranes and intentionally-defined membranes (using pointcuts to specify the boundaries).

5. Putting membranes in perspective

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 [15], 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 should be able to use bisimulation techniques at the level of membranes, although this is future work. A fundamental difference between open modules and membranes is that membranes can be dynamic. EffectiveAdvice [8] 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.

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. Second, generalizing, an advising membrane can 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 [9, 11]. 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.

6. Conclusion

To conclude, we believe that adapting the notion of programmable membranes in order to control the controversial power of aspects is a promising direction. Membranes support flexible topological scoping with locally programmable weaving rules. Much work remains to be done to unleash the potential of membranes for modular reasoning, encapsulation, security, and scoping of aspects, including in presence of concurrency and distribution.

Specifically, a major challenge lies in supporting crosscutting membranes properly. On the one hand, dynamic membrane deployment—as supported by both MAScheme and PHANtom—makes crosscutting inevitable. On the other hand, unrestricted crosscutting between membranes can destroy the guarantees that one seeks with membranes, such as avoiding infinite loops and enforcing encapsulation policies. Addressing this tension is necessary to find a proper design for membrane-based aspect languages.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. Black, editor, *ECOOP 2005*, number 3586 in LNCS, pages 144–168, Glasgow, UK, July 2005. Springer-Verlag.

[2] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In *AOSD 2011*, pages 141–152, Porto de Galinhas, Brazil, Mar. 2011. ACM Press.

[3] G. Boudol. A generic membrane model (note). In *Global Computing Workshop*, vol. 3267 of *LNCS*, pages 208–222. Springer-Verlag, 2005.

[4] 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.

[5] J. Fabry and D. Galdames. PHANtom: a modern aspect language for Pharo Smalltalk. In *International Workshop on Smalltalk Technologies*. ACM Press, 2011. To Appear.

[6] 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.

[7] M. Inostroza, É. Tanter, and E. Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *ESEC/FSE 2011, New Ideas track*, pages 508–511, Szeged, Hungary, Sept. 2011.

[8] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In *AOSD 2010*, pages 109–120, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[9] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *ECOOP 2008*, number 5142 in LNCS, pages 155–179, Paphos, Cyprus, july 2008. Springer-Verlag.

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

[11] 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):1, 2010.

[12] É. Tanter. Expressive scoping of dynamically-deployed aspects. In *AOSD 2008*, pages 168–179, Brussels, Belgium, Apr. 2008.

[13] É. Tanter. Execution levels for aspect-oriented programming. In *AOSD 2010*, pages 37–48, Rennes and Saint Malo, France, Mar. 2010.

[14] É. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *GPCE 2010*, pages 113–122, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

[15] É. Tanter, N. Tabareau, and R. Douence. Exploring membranes for controlling aspects. Technical Report TR/DCC-2011-8, University of Chile, June 2011.