

Analyzing Interactions of Structural Aspects

Benoit Kessler Éric Tanter*

DCC, University of Chile
Avenida Blanco Encalada 2120
Santiago, Chile
{bkessler, etanter}@dcc.uchile.cl

Abstract. A structural aspect is an aspect whose action consists in changing the structure of a program element, for instance by adding a method to a class. So far, very little has been done in the community to address the issues arising from interactions of structural aspects. In this paper, we present a classification of potential interactions of structural aspects, and highlight possible resolution schemes for such interactions. We are now working on a comprehensive solution to these issues in the context of Reflex, a versatile kernel for multi-language AOP in Java. For the detection of these interactions, we are exploring the use of a logic engine, capable of reporting subtle interactions between structural aspects.

1 Introduction

Although most of AOP approaches focus on *behavioral* aspects following the pointcut-advice model of AspectJ [5], *structural* aspects, as exemplified by inter-type declarations (*aka.* introductions) in AspectJ, seem to find quite a number of applications in real cases. A structural aspect is one that, as part of its action, modifies the structure of program elements. Mostly, structural aspects in current proposals are able to *add* members or interfaces to classes, and to change the class hierarchy.

As this kind of aspects becomes popular, the case of their interactions turns out to be crucial. Although most work in the community related to aspect interactions only deals with behavioral aspects, some proposals are starting to emerge that deal with structural aspects [4, 7]. The basic idea is that since aspects mainly base their cut on structural properties of the program (possibly augmented with dynamic properties), the fact that some aspects may alter this structure can result in inconsistencies and surprises due to (hidden) dependencies.

In this paper, we propose a first analysis of the possible interactions of structural aspects *adding* structural elements, and sketch a general approach for their automatic detection, while highlighting desirable resolution schemes. We do not explicitly deal with the dependencies between structural and behavioral aspects

* É. Tanter is financed by the Milenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

here (elements can be found in [7]), nor do we focus on changes to class hierarchies for now.

2 Possible Interactions and their Resolution

In this section, we propose a classification of structural interactions, and list the possible resolutions that one would expect from a comprehensive system truly supporting structural aspects. We distinguish conflicts occurring from the interaction of an aspect with the original base code, conflicts between actions of (at least) two aspects, and dependencies between the action of an aspect and the cut of another one. For each category, we give a general description of the syndrome, a few examples, and a list of desired possibilities of resolution. Note that every suggested possibility is a priori always applicable.

In the following, a *structural element* denotes any piece of structure in an OO program, *i.e.* a class, interface, annotation, field, method or constructor. The actions we consider are the *addition* of structural elements to the base program, *e.g.* a new class, a new method to a class, or a new annotation to a field. A *structural container* is an element containing other structural elements; for instance, the VM is a structural container of classes, a class is a structural container of members, and a member is a structural container of its annotations.

2.1 Interactions with Base Code

Syndrome	An aspect adds a structural element which is already present in the base code.
Examples	Add class C but C already exists. Add method m to class C which already has this method (either directly or via inheritance).
Treatments	Skip the action. Combine element to add with existing one (*). Modify the aspect to avoid the clash.

(*) By *combining*, we are referring to a mechanism similar to what is proposed in composition of traits [6]: explicit aliasing of conflicting members and definition of a combination based on the aliased members.

2.2 Interactions between Actions

Syndrome	Two aspects add an element with the same signature in the same structural container.
Examples	Aspects A1 and A2 add a class C . Aspects A1 and A2 add a method m to class C (either directly or via inheritance).
Treatments	Skip one or both of the actions. Combine both elements to add in a single one (*). Modify one or both aspects to avoid the clash.

2.3 Interactions Action-Cut

Syndrome	An aspect adds an element which belongs to the intensional cut of another aspect.
Examples	Aspect A1 adds a class C to package p , and aspect A2 adds a method m to all classes of p . Aspect A1 adds an annotation to all fields of class C , and aspect A2 adds a field to class C .
Treatments	Make added element visible or not to (the cut of) other aspects. Control order of application of aspects.

3 Towards Automatic Detection of Structural Interactions

Admittedly, a crucial part in aspect composition lies in the *detection* of the interactions [3]. As a matter of fact, aspects typically use *intensional* definitions of their cuts, so the developer can have trouble foreseeing the possible interactions between a given aspect and the base code, and between several aspects. It is therefore crucial that the AO system *detects* and *reports* on interactions.

Some interactions (action-base and action-action in our classification) are easy to detect since they result in compilation errors (*e.g.* a class with two methods of the same signature). AspectJ for instance reports on this class of conflicts, since the compilation process cannot go any further.

More subtle is the case of action-cut interactions, because there is no compilation error implied, nor are the dependencies between aspects easy to see – in particular if the cut and action languages are Turing-complete. In order to detect these interactions, we must be able to reason from two sets of pieces of information: which aspects *look at* which structural elements for their cut and action? which aspects *change* which structural elements as a result of their action?

3.1 An interactive composition process

In order to give the programmer the necessary information to compose the aspects together, we require a mechanism to detect and report the interactions between aspects. In our proposal, this detection mechanism is integrated in a wider process for aspect composition. This process aims at presenting the detected interactions according to the specifications of the programmer. The programmer can specify rules of ordering, visibility, dependencies (*e.g.* mutual exclusion), etc., in the line of what is proposed in [7]. The issue of how these specifications affect the detection mechanism is not dealt with in this paper.

3.2 A logic-based approach

The approach we are currently exploring is the use of a *logic engine* connected to our AOP platform supporting structural aspects (Reflex [8, 7]). This is feasible and interesting to do, as explained in [2], because this part of the system is

strongly logic-oriented and including logic programs has become efficient enough. The current implementation of detection and resolution of interactions between behavioral aspects in Reflex is purely Java-based, and the obstacles we encountered strongly motivated this change of implementation approach.

As any logic engine, our system uses rules which reason on a set of facts. These facts represent what is done within Reflex and are generated as explained in the next section.

3.3 Rules and facts

Fact generation. Facts representing what happens in Reflex and on the classes of the program being manipulated are generated at different levels:

- Upon introspection, entities from our structural model (classes, methods, fields, etc.) generate logic facts indicating that they are being observed by a given link. For instance, link `l1` cuts the classes which have a field with the annotation `@Persistent`. It first accesses the pool¹ of fields of the class and then, on each field it accesses the pool of annotations of the field and finally, on each annotation, it reads its name. Each structural element generates a fact when it is being accessed and therefore the class generates the fact that `l1` reads its pool of fields (`readFields('l1','C')`), each field generates the fact that `l1` reads its pool of annotations (`readFieldAnnotations('l1','C','f')`) and each annotation generates the fact that `l1` reads its name (`readFieldAnnotationName('l1','C','f')`).
- Upon intercession (*i.e.* structural changes), structural elements generate logic facts indicating the changes being made. For instance, class `C` generates the fact that link `l2`, applied on class `C`, adds the annotation `@Persistent` on the field named `f`, in class `C2` (`C` and `C2` may not necessarily be the same). That is `addAnnotationToField('l2','C','C2','Persistent','f')`.
- Upon specification of rules of ordering, visibility or dependency, additional facts are generated. These facts are not directly involved in the interactions, but they influence the detection as they determine the scope of the aspects and the feasibility of the interactions.

Accuracy of detection vs. expressiveness of the cut language. Regarding all the facts generated during the introspection, the question of the accuracy of the detection naturally arises. In fact, there is trade off between expressiveness of the cut language and the accuracy of the detection². In Compose* for

¹ We say that each class has a *pool* of fields, a pool of methods, a pool of constructors, etc. Similarly, other structural elements may have pools of other elements, such as annotations.

² The action language does not cause any problem as, in Reflex, we precisely know what changes are effectively done since structural entities know what changes are performed upon them, as well as by which link. So the facts generated for structural changes are accurate and always correct.

instance, the cut language has limited expressiveness (it is not Turing-complete) and greatly resembles a logic language. Hence with such a language it would be straightforward to ensure that only appropriate facts are generated. On the other hand, this means not being able to express advanced selection criteria, *e.g.*: cut every class which has a method with a specific annotation, but no more than three annotations, exactly four parameters and two of type `int`.

The path we choose is to keep the expressiveness given by our reflection-based model (*i.e.* cut and action are expressed as Java methods manipulating reifications of the structural elements). Then there are two alternatives: (a) the structural elements automatically generate logic facts as they are being observed, (b) the user explicitly specifies (*e.g.* as annotations or using an embedded DSL) what the cut does.

In the first case, there is a possibility that too many facts be generated. For instance, in the example introduced in section 3.3, only the fact that `11` reads the name of the annotation on a field is relevant as it was the intention of the programmer. The two other facts may lead to non-existent interaction detection as they get involved with other rules. Therefore we detect spurious conflicts, but this is arguably better than missing effective conflicts. In the second case, an explicit contract is actually expressed by the aspect programmer, and hence we can generate the facts that precisely correspond to the intention of the programmer. On the other hand, it is the burden of the programmer to declare this contract. At present we are rather exploring the first alternative.

Rules. Along with the three kinds of facts generated, we define rules for detecting interactions. These rules follow the principle of section 2.3: they look for actions interacting with cuts. Below is an example of the minimal rule which detects all interactions between aspects that adds an interface to a class and aspects that cuts classes with that interface:

```
interactInterfaceName(Link1,Link2,Class1,Class2,InterfaceName)
:- readClassInterfaceName(Link1,Class1,InterfaceName),
   addInterface(Link2,Class2,Class1,InterfaceName).
```

`Class2` is the class on which `Link2` is applied and `Class1` is the class where `Link2` adds the interface with name `InterfaceName`. The result of the detection using this rule is a tuple which is to be understood as: "*Link2 interacts with Link1 because Link2, applied to class Class2, adds the interface named InterfaceName to class Class1, while Link1 is looking for that interface on class Class1.*"

An interaction rule can also take into account aspect *ordering* specifications and/or *visibility* of structural changes [7] in order to report or not an interaction:

```
interactInterfaceName(Link1,Link2,Class,Class2,InterfaceName)
:- readClassInterfaceName(Link1,Class,InterfaceName),
   visible(Link2,Link1).
   addInterface(Link2,Class2,Class,InterfaceName).
```

The above rule extends the previous one in order to take into account the fact that modifications done by `Link2` are visible to the cut of `Link1`, otherwise no interaction is detected.

4 Further Issues

There are a number of issues that we explicitly do not address in this preliminary work. First of all, structural aspects typically have the possibility to alter the class hierarchy, by changing the superclass relation of a class. These changes introduce new kinds of dependencies, which ought to be studied further.

Also, even if structural aspects are usually perceived as doing either class hierarchy changes or additions of structural elements, the underlying abstraction layer of structural reflection gives far more powerful changes, *i.e.* by actually supporting *modifications* of elements, like renaming them or changing their modifiers. A first question is if those changes make sense in the restricted context of AOP, and if so, how to detect and resolve the new interactions that can appear.

Finally, our stance on the case of *circular dependencies* differs from the approach of `Compose*` [4]. In the face of a circular dependency between two aspects, changing the relative order of application produces two different results. Our approach consists in informing the programmer of all the possible interactions and of the existence of circular dependencies and to let the user decide of the ordering of the aspects (globally or for each class), rather than rejecting the specification.

5 Related Work

As of now, there has been very little work on the composition issue between structural aspects. Most work on aspect composition focuses on behavioral aspects. In `AspectJ` [5], base-action and action-action conflicts are reported as compilation errors, while action-cut conflicts are not reported. Furthermore, very little expressive power is given to the programmer to resolve conflicts. In `Compose*` [4], the approach consists in trying to automatically order structural actions properly, and reject any specification that leads to circularity. The automatic approach to resolution of interactions is interesting, but we rather share the point of view that resolution should be done explicitly, as in many cases, the precise resolution depends on particularities of the considered application [3]. Finally, in related areas dealing with structural composition, we find the traits approach very promising for its flexibility and expressive power given to the programmer [6]: when a class uses two traits that both have the same method, the programmer is informed and can define a specific alias for each trait method and combine them in a new method added to the class.

6 Conclusion

We have presented a classification of potential interactions in structural aspects, and highlighted possible resolution schemes for such interactions. We are now

working on a comprehensive solution to these issues in the context of Reflex. With respect to resolution of interactions, Reflex already supports most of what we need: operators for ordering of aspects, mutual exclusion between aspects, and a mechanism to support subjective *views* on the program structure [7]. Combinators *a la traits* are still lacking but the underlying infrastructure (Javassist [1]) is flexible enough to support them. The most challenging part concerns the *detection* of interactions. As we have sketched in this paper, we are studying the use of a logic engine from within Reflex to achieve this.

References

- [1] S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.
- [2] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems – A case study. *ACM SIGPLAN Notices*, 31(5):117–126, May 1996.
- [3] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of Lecture Notes in Computer Science, pages 173–188, Pittsburgh, PA, USA, Oct. 2002. Springer-Verlag.
- [4] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 214–225, Bonn, Germany, Mar. 2006. ACM Press.
- [5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.
- [6] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In L. Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 248–274, Darmstadt, Germany, July 2003. Springer-Verlag.
- [7] É. Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, Lecture Notes in Computer Science, Vienna, Austria, Mar. 2006. Springer-Verlag. To appear.
- [8] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of Lecture Notes in Computer Science, pages 173–188, Tallinn, Estonia, Sept./Oct. 2005. Springer-Verlag.