Flexible Model Element Introduction Policies for Aspect-Oriented Modeling[⋆]

Brice Morin[1,2], Jacques Klein[3], Jörg Kienzle[4], and Jean-Marc Jézéquel[1,5]

[1] INRIA, Centre Rennes - Bretagne Atlantique, France
Brice.Morin@inria.fr
[2] SINTEF ICT, Oslo, Norway
[3] Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
Jacques.Klein@uni.lu
[4] School of Computer Science, McGill University
Joerg.Kienzle@mcgill.ca
[5] IRISA, Université de Rennes1, France
jezequel@irisa.fr

Abstract. Aspect-Oriented Modeling techniques make it possible to use model transformation to achieve advanced separation of concerns within models. Applying aspects that introduce model elements into a base model in the context of large, potentially composite models is nevertheless tricky: when a pointcut model matches several join points within the base model, it is not clear whether the introduced element should be instantiated once for each match, once within each composite, once for the whole model, or based on a more elaborate criteria. This paper argues that in order to enable a modeler to write semantically correct aspects for large, composite models, an aspect weaver must support a flexible instantiation policy for model element introduction. Example models highlighting the need for such a mechanism are shown, and details of how such policies can be implemented are presented.

1 Introduction

Abstraction and separation of concerns are two fundamental principles applied in software engineering to address a continuously wider range of software development problems. In particular, abstraction and separation of concerns can help to deal with the increasing complexity of modern software.

The Model-Driven Engineering (MDE) paradigm [17] proposes to consider models as first-class entities. A model captures a given aspect of the reality, for example the structure (class or component diagrams) or the behaviour (state machines or sequence diagrams) of a system. Even if a model is an abstraction of the reality (simpler than the reality) and hides irrelevant details, modelling a

[⋆] This work was partially funded by the DiVA project (EU FP7 STREP, contract 215412, http://www.ict-diva.eu/), and the SPLIT project (FNR and CNRS funding, http://wiki.lassy.uni.lu/Projects/SPLIT), as well as the Canadian National Sciences and Engineering Research Council.

complex system is not a simple task and the size of a model of an application can grow rapidly, thus reducing its maintainability, readability, etc.

The Aspect-Oriented Modelling (AOM) paradigm [19, 1, 10], which is inspired by AOP [8, 9], mixins [18] and Feature-Oriented Programming [16], is an attempt towards a better separation of concerns in MDE. Over the last decade, many AOM approaches have been proposed with very few publicly available (or maintained) implementations. Noticeable exceptions are the Motorola WEAVR [2], dedicated to the weaving of AspectJ-like aspects into state machines, or the academic weavers implemented in Kermeta [15] by the Triskell team (and associated teams): Kompose[6] [3] (with Colorado State University), GeKo[7] [12] (with University of Luxembourg) and SmartAdapters[8] [13] (Formerly with I3S Nice-Sophia Antipolis and Institut Telecom/Université de Lille 1, recently improved within the DiVA project). Because of the lack of functioning weavers, many of the proposed approaches have only been applied to simple academic examples and were not tested on real-life models. In particular, these approaches have difficulties to weave aspects into composite or hierarchical models (class diagrams with packages, sub-packages and classes ; composite state machines ; composite component diagrams ; etc). Indeed, they often only offer one way for an aspect model to introduce new model elements into the base model.

This paper argues that in order to enable a modeller to write semantically correct aspects for large, composite models, an aspect weaver must support a flexible instantiation policy for model element introduction. Section 2 motivates the need for such a mechanism by example. Section 3 explains that the desired flexibility can be added to existing AOM approaches by augmenting the weaver with an *advice sharing* capability. For every model element introduced by an aspect, the modeller can specify if and how instances of the element are shared between multiple applications of the advice model. Implementation details of advice sharing in the context of the *SmartAdapter* approach are presented in Section 4. Section 5 shows how advice sharing can be used in a real-world model to implement state recovery. Secton 6 presents related work, and the last section draws our conclusions.

2 Motivating Example: A (not so) Simple Log Aspect

We propose to illustrate the need for advice sharing in the context of hierarchical component diagrams. The example aspect that we are going to use is a simple *Logging* aspect presented in Figure 1. The effect of this aspect model consists in introducing a new *Logger* component into the model, and linking any component that requires the log service to it.

In the following we illustrate how the result of applying the *Logging* aspect to different base models using different weaving policies can produce drastically different results. If we apply the *Logging* aspect on a simple flat component architecture, a new *Logger* is introduced each time a match of the pointcut model

[6] https://gforge.inria.fr/frs/?group_id=32
[7] http://se2c.uni.lu/tiki-index.php?page=Geko
[8] http://divastudio.gforge.inria.fr/bundle/latest_build/DiVAStudio.zip

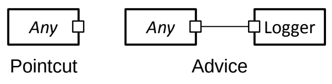

Fig. 1. A Simple Logging Aspect

(in this case a component *Any*) is detected in the base model. As a result, each component that requires the log service is going to be connected to its own logger, as illustrated in Fig. 2.

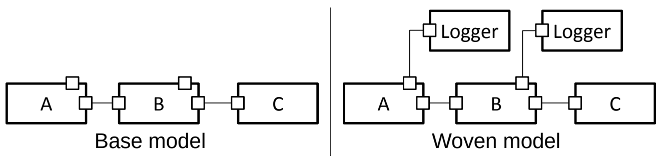

Fig. 2. Weaving the Simple Logging Aspect Into a Flat Base

This result is probably not the one we would like to achieve. Rather, we would like to obtain the result illustrated in Figure 3. In this case, all the components that require the log service are connected to the same *Logger* instance. In other words, the aspect only introduces *one unique* instance of the *Logger* component. However, the link that is created between the component and the *Logger* instance is created *per pointcut match*. We introduced the original notion of *uniqueness* for aspect-oriented modeling in 2007 in the context of the *SmartAdapters* approach [14]. In 2009, Grønmo *et al.* [6] introduce a collection operator for graph transformation, which achieves a similar effect, thus confirming the need for more elaborate introduction control. In the following we will call this introduction strategy *global*, instead of *uniqueness*.

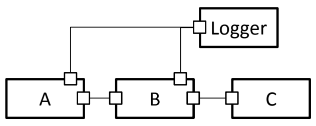

Fig. 3. Weaving the Simple Logging Aspect: Expected result.

Let us now consider an architecture composed of a hierarchy of components, such as the one illustrated in Figure 4.

In this case, we would like the application of our *Logging* aspect to result in the creation of a *Logger* component for each composite component. The *Logger*

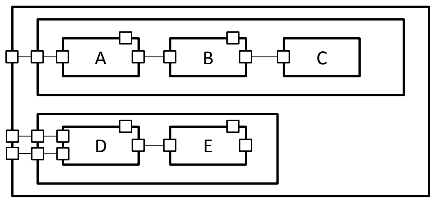

Fig. 4. A Composite Architecture

instance of the composite component would then be used by all the internal components of the composite. This strategy is useful in order to log warnings, errors, etc. separately for each composite with the aim of improving readability.

Without a more fine-grained introduction support, we cannot write a simple *Logging* aspect that would achieve the desired effect. If we do not declare the introduced *Logger* component as global, we end up with 4 new *Logger* instances connected to A, B, D and E. We can not declare the introduced *Logger* component as *global*, because we want to introduce more than one *Logger* instance. Also, since a single model element cannot be contained by multiple containers (the subcomponent relationship is a composite reference in the architectural metamodel), it is not possible to add the same logger component in different composite components. More precisely, the logger component will be added into the first composite component and bound to all the internal components, i.e. A and B. Next, it will be moved inside the second composite component, and bindings are established to C and D. At the end, this would lead to an erroneous configuration, with bindings that "cut across" the boundaries of their composite components as illustrated in Figure 5.

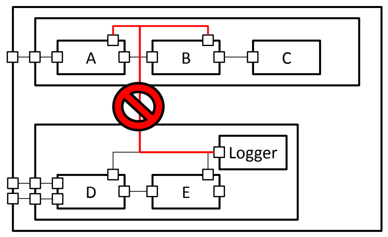

Fig. 5. A Composite Architecture, with the Logging Aspect Badly Woven

We hope that this simple example highlights that there is a real need for a more flexible notion of model element introduction in AOM. The advice model should be able to specify how model elements that are introduced into the base model are shared between multiple join points. Note that the problem we are describing here is common to all the pointcut-based AOM weavers, including GeKo [12], MATA [19, 7], and XWeave [5]. This problem is not specific to composite architectural models, as we can find the problem in hierarchical state machines, class diagrams (with packages and sub-packages), or even sequence diagrams (see Section 5). In our *Logger* example, using a more flexible notion of *sharing* of model elements introduced by an advice, it should be possible to obtain the desired result illustrated in Figure 6.

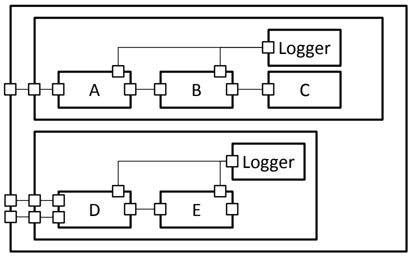

Fig. 6. A Composite Architecture, with the Logging Aspect correctly woven.

3 Flexible Policies for Sharing Introduced Model Elements between Multiple Join Points

3.1 A Simple Metamodel for Pointcut-based Weavers

In pointcut-based approaches (*e.g.* SmartAdapters), an aspect is composed of three parts, as illustrated by Figure 7:

1. an advice model, representing **what** we want to weave,
2. a pointcut model, representing **where** we want to weave the aspect and
3. weaving directives specifying **how** to weave the advice model at the join points matching the pointcut model.

Both the advice and the pointcut models allow the designer to define the aspect in a declarative way (*what*), whereas the weaving directives allow the designer to specify the composition. This can be done using a simple statically-typed imperative (*how*) language (like in SmartAdapters [14]), with graphical composition directives (like in MATA [19] using `<<create>>` and `<<delete>>` stereotypes) or by specifying mappings (like in GeKo [12]).

3.2 A Metamodel for Sharing of Introduced Model Elements

The behavior of weavers in general is to weave the model elements that are introduced in an advice model into the base model each time the pointcut model matches. In [11] we had already introduced the notion of *uniqueness* for introduced advice model elements as part of the SmartAdapters approach. Unique

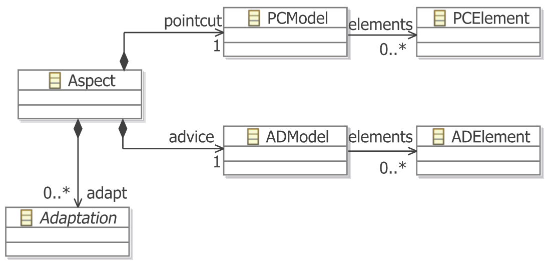

Fig. 7. The SmartAdapters Core Metamodel

elements are instantiated only once, making it possible to reuse these elements during each composition of an aspect within the same base model. However, as shown by example in section 2, this policy is not sufficient to handle models with hierarchy, e.g., state charts with composite states, component diagrams with composite components, class diagrams with packages and sub-packages, etc. Diagrams with object instances such as sequence diagrams or communication diagrams are also problematic, as shown later in section 5.

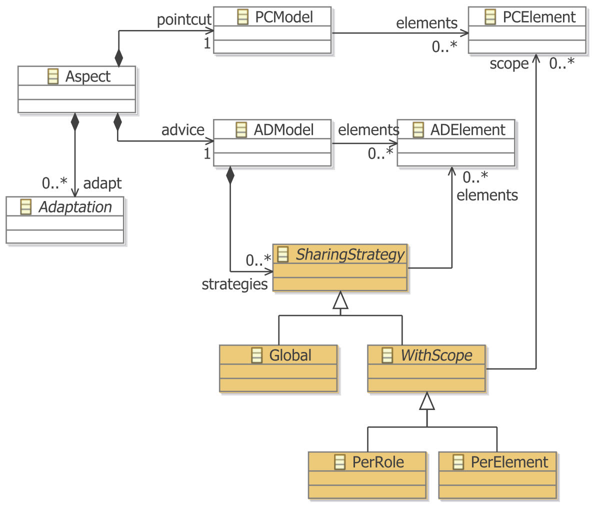

Fig. 8. Extended SmartAdapters Metamodel for Handling Introduction Strategies

The main contribution of this paper is to introduce the notion of *sharing of introduced model elements* into aspect-oriented modeling. To this aim, the advice model must conform to the extended metamodel MM' shown in Fig. 8. We propose to associate an instantiation strategy to model elements introduced in an advice model. In the case the weaver determines several join points in the base model that match the pointcut model, the designer can choose among several instantiation strategies for each introduced model element in the advice model:

- **Per Pointcut Match**. A new instance of the advice element is introduced for each pointcut match. See for example the bindings in Fig. 2. This is the standard behavior of conventional model weavers.
- **Global**. A single instance of the model element is introduced into the base model, regardless of how many join point matches are found. In the case where there are several join points, the model elements designated as global are introduced only the first time the advice is applied, and reused in the subsequent applications of the advice. The *Logger* component in Figure 3 is an example of a global element.
- **Per Role Match** $(pe_1, pe_2, ..., pe_n)$. An instance of the element is introduced each time the pattern matcher associates a different *tuple* of base elements $(be_1, be_2, ..., be_n)$ with the model elements $(pe_1, pe_2, ..., pe_n)$ in the pointcut model. In other words, we reuse the same instance of an introduced model element in all the weavings where base model element be_i plays the role pe_i defined in the pointcut model.
- **Per Element Match** $(pe_1, pe_2, ..., pe_n)$. An instance of the element is introduced each time the pattern matcher associates a different *set* of base elements be_i with the model elements pe_i in the pointcut model. In other words, we reuse the same instance of an introduced model element in all the weavings where the set of base model elements be_i that is matched by pe_i is the same, regardless if the actual role the base elements play remains the same or not.

To illustrate the difference between *per role match* and *per element match*, let us introduce a very simple example, illustrated in Figure 9. The pointcut model is composed of two (symmetric) connected components *Any* and *Another*. The advice model simply links a component X to these two components. In the base model composed of two connected component A and B, the pointcut matches twice: (1) *Any*→*A*, *Another*→*B*, and (2) *Any*→*B*, *Another*→*A*. In the case of the *PerElementMatch(Any, Another)* strategy, a single instance of X is introduced, since both pointcut matches globally maps to the same set of model elements. However, in the case of the *PerRoleMatch(Any, Another)*, X is instantiated twice because in the two pointcut matches, A and B play different roles.

In hierarchical models the *PerRoleMatch* and *PerElementMatch* can be used to define a *scope* for a weaving as illustrated in the new version of the *Logging* aspect shown in Fig. 10. We added in the pointcut the composite component *Container* containing the component that requires the log service. This composite allows us to define the scope of the *Logger* component and associate the logger component with a *PerRoleMatch(Container)* strategy[9] to achieve the weaving illustrated in Fig. 6. Indeed, in this way, the Logger component will be only instantiated once for all pointcut matches in a same Container.

[9] Note that in the case where only one model element is given in the introduction policy like it is the case in this example, *PerRoleMatch* and *PerElementMatch* yield identical results.

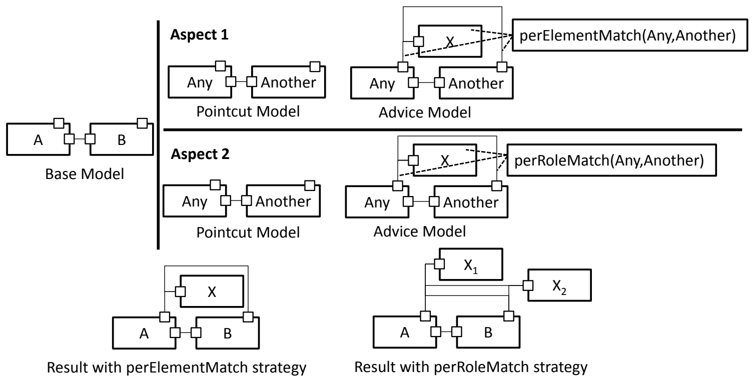
Aspect 1
perElementMatch(Any,Another)
X
Any
Another
Pointcut Model
Advice Model
A
B
Base Model
Aspect 2
perRoleMatch(Any,Another)
Result with perElementMatch strategy
Result with perRoleMatch strategy

Fig. 9. Per Role Match and Per Element Match strategies

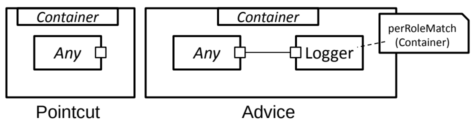

Fig. 10. Logging Aspect revisited with a PerRoleMatch Introduction Strategy

4 Implementation of Advice Sharing

This section provides implementation details on how we implemented the flexible introduction policies described in section 3 in the context of SmartAdapters. However, we believe that our description is general enough to provide insight to readers wishing to integrate the same strategies to other approaches.

In our latest version of SmartAdapters, our weaver generates for each aspect model Drools and Java code to perform the weaving. Drools (*a.k.a* JBoss Rules) is a rule engine that implements the RETE algorithm [4] with optimization for object-orientation to efficiently find patterns in OO structures, and is used in SmartAdapters to perform join point detection. The Java code performs the actual weaving of the advice for each join point match. The use of Drools and Java is however completely hidden from the modeler. All scripts presented in this section were automatically generated by our weaver[10].

The generated script for the compiled version of the *Logging* aspect where the *Logger* component uses a global introduction policy (see Fig. 1 and Fig. 3) is shown below. In Drools, each application of a rule is independent from the previous ones. For example, if the pointcut (*when clause* lines 3 to 7) matches several times in a base model, it is not possible to directly know if an element from the advice (*then clause* lines 8 to 21) has already been created in a previous application of the

[10] The only modifications made to the generated scripts to improve readability are: the renaming of some variables, the introduction of comments, the removal of casts and the removal of the lines not relevant for the discussion.

aspect. This is not a problem for the standard *PerPointcutMatch* strategy, but becomes an issue that must be overcome in order to implement the other three introduction policies.

Luckily it is possible to declare global variables, initialized by the Java code calling the Drools engine with the script, to share data between different application of the advice. We thus use the `globalElem` map (see line 1) as the structure that keeps track of the global elements of the advice model. Basically, each advice element is identified by a unique string. Before creating a unique element, we check if this element already exists in the `globalElem` global map (see line 12 and 13). It this element already exists, it is reused. Otherwise, it is created and stored in the global map (lines 14 and 15).

```
 1  global Map<String, EObject> globalElem;
 2  rule "LoggingAspect"
 3  when //Pointcut
 4    $logService: type.Service(name=="org.slf4j.Logger")
 5    $reqLogPort: type.Port(role=="client", service==$logService)
 6    $anyType: type.PrimitiveType(port contains $reqLogPort)
 7    $anyCpt: instance.PrimitiveInstance(type==$anyType)
 8  then
 9    //Creation of Advice model elements
10    ...
11    //DefaultLogger component (unique)
12    instance.PrimitiveInstance logCpt = globalElem.get("logCpt");
13    if (logCpt == null){
14      logCpt = createPrimitiveInstance();
15      globalElem.put("logCpt",logCpt);
16    }
17    ...
18    //Binding to the DefaultLogger component (per pointcut match)
19    instance.Binding binding = createBinding();
20    ...
21  end
```

The following script is the compiled version of the *Logging* aspect when the *Logger* component uses the *PerElementMatch(Container)* instantiation policy (see Fig. 10). We also use a global map `perElem`; however, the key of this map is a set of `EObject` (model elements) from the base model. In Java, the hash code of a set is computed as the sum of all the hash codes of the elements contained in this set. It thus can be possible that two different sets (containing different elements) have the same hash code. While this is not recommended (for performance issues), a Map can handle this case and retrieve the right values even if two (different) keys have the same hash code. Indeed, according to the Java specification, the `containsKey(Object key)` method "returns true if and only if this map contains a mapping for a key k such that (`key==null ? k==null : key.equals(k)`)". Finally two sets are equals if "the two sets have the same size, and every member of the specified set is contained in this set (or equivalently, every member of this set is contained in the specified set).” This is exactly the behavior we need to implement the *PerElementMatch* introduction policy.

```
global Map<Set<EObject>,Map<String, EObject>> perElem;
rule "LoggingAspect"
when //Pointcut
  $logService: type.Service(name=="org.slf4j.Logger")
  $reqLogPort: type.Port(role=="client", service==$logService)
  $anyType: type.PrimitiveType(port contains $reqLogPort)
  $anyCpt: instance.PrimitiveInstance(type==$anyType)
  $anyComposite: instance.CompositeInstance(subCpts contains
      $anyCpt)
then
  //Init of the structure managing the per element strategy
  Set<EObject> compositeScope = new HashSet<EObject>();
  compositeScope.add($anyComposite);
  if (perElem.get(compositeScope) == null){
    perElem.put(compositeScope,
          new Hashtable<String, EObject>());
  }
  //Creation of Advice model elements
  ...
  //DefaultLogger component (unique with scope)
  instance.PrimitiveInstance logComponent =
    perElem.get(compositeScope).get("logComponent");
  if (logComponent == null){
    logComponent = createPrimitiveInstance();
    perElem.get(compositeScope).put("logComponent",logComponent);
  }
  ...
end
```

In the same way, the *perRoleMatch* mechanism can be implemented with a Map<Map<String, EObject>, Map<String, EObject>>, where:

- the key is a Map<String, EObject>, where:
  - the key is a String identifying the role of a pointcut model element
  - the value is a base model element that matches the role
- the value is a Map<String, EObject>, where:
  - the key is a String identifying an advice model element
  - the value is a clone of an advice model element

5 Using Flexible Introduction Policies to Implement Recovery

This section demonstrates by means of a real-world example how important it is for a weaver to support flexible sharing of introduced model elements.

In data-centric dependable systems, the consistency of the state of the application is of utmost importance. Any modification of the application state is carefully checked to make sure that no inconsistencies are introduced. In the case where an error is detected, the application must perform recovery actions in order to address the situation. To restore application consistency, *forward* or *backward error recovery* techniques can be used.

Forward error recovery consists in first performing detailed damage assessment to determine which parts of the application state are inconsistent. Then, additional operations on the erroneous application state are performed to create a (new) consistent state. In backward error recovery, the application state is saved periodically. When an error is detected, the application is returned to that (old) consistent state by undoing all the state changes performed since the last save point.

Forward and backward error recovery can be implemented using different techniques. One popular way is to keep track of all operation invocations on application data objects at run-time. This information is useful for detailed damage assessment (used in forward error recovery), as well as for undoing state changes by executing compensating actions (useful for backward error recovery).

To implement operation tracing, an operation list is usually associated with each data object. Each time a modifying operation is invoked on the data object, the operation is saved in the list. In addition, each data object that is modified is registered with a recovery manager, i.e. with an object that coordinates the recovery process of all affected data objects in case of a failure.

This behaviour can conveniently be described using a sequence diagram aspect model as shown in Fig. 11. The pointcut model states that we are looking for places where a `caller` instance of class `C` synchronously invokes a method `m` on a `target` instance of class `Data`. In this case the `Data` class encapsulates the valuable application data that needs to be kept consistent. It is obvious that in any real-world base model, a weaver would most likely find matches for such a pointcut. The advice model of the aspect in Fig. 11 states that several *new* model

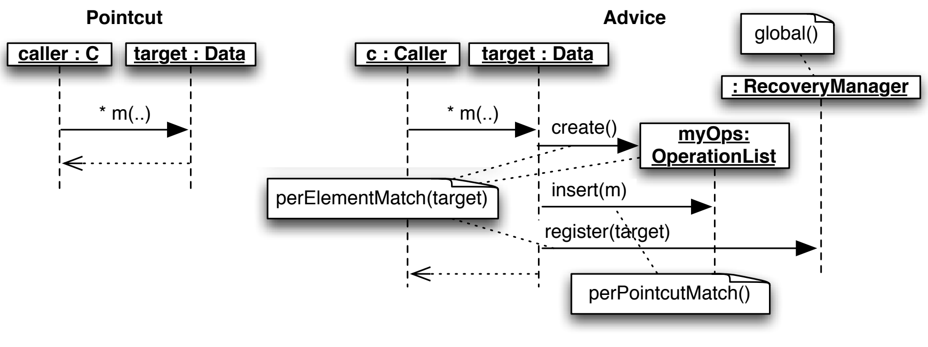

Fig. 11. A Sequence Diagram Aspect Model for Recovery Support

elements are to be introduced into the execution of the method call `m`:

1. a constructor call to `create`, which instantiates
2. a new `myOps` instance of the class `OperationList`
3. a call to the method `insert` of `myOps`
4. an unnamed instance of the `RecoveryManager` class
5. a call to the method `register` of the `RecoveryManager` instance

In general, there is only one recovery manager per application, therefore the introduction of the `RecoveryManager` instance must be tagged as *global*. Every instance of the class `Data` that is the target of a method call must get its own associated `OperationList` instance to store the method invocation (and all future subsequent ones). Hence the `myOps` instance introduction is tagged as *perElementMatch(target)*. As a result, the constructor call introduction must also be tagged as *perElementMatch(target)*. The `insert` method call introduction, however, is tagged as *perPointcutMatch*, since each and every method invocation needs to be stored for recovery to be possible. Finally, each instance of the class `Data` that is accessed needs to register with the recovery manager. To make this happen only once for each `Data` object, the call to the `register` method is tagged *perElementMatch(target)*.

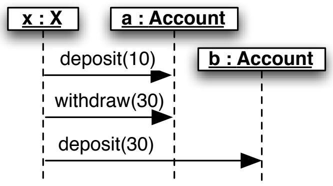

Fig. 12. A Banking Base

Applying the aspect of Fig. 11 to a simple banking base model such as the one shown in Fig. 12 results in the woven model shown in Fig. 13. The result is as desired: there is only 1 instance of `RecoveryManager`, there are two `OperationList` instances, two calls to `create`, two calls to `register`, and 3 calls to `insert`.

6 Related Work

To the best of our knowledge, the current existing AOM weavers using a pointcut-based approach do not propose sophisticated solutions to deal with the problem of model element introduction in the presence of multiple join points. For instance, after a phase of detection of join points corresponding to a pointcut model, SmartAdapter [13] uses adaptations to compose an aspect model with a base model. But currently, even if these adaptations are fine grained, they only allow the specification of global or per pointcut match strategies [11]. Approaches such as MATA [19], XWeave [5] or GeKo [12] do not propose solution for the presented

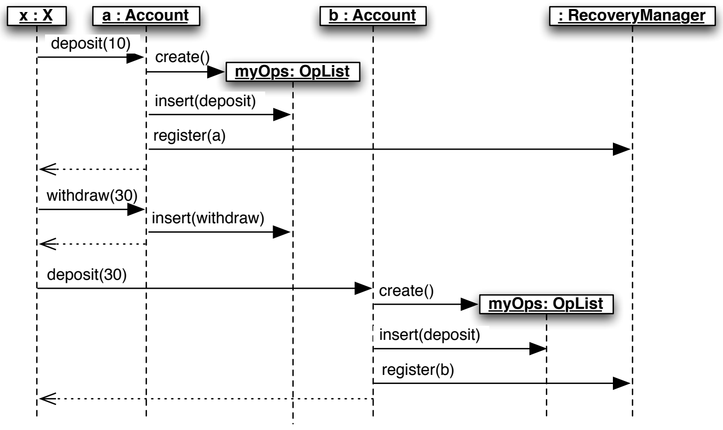

Fig. 13. The Result of the Weaving (3 Matches)

problem. MATA is based on graph transformations. The base model is converted into a graph. The aspect model that is to be woven is converted into a graph transformation rule, and the rule is executed on the base graph. XWeave is an asymmetric AOM approach based on the *Eclipse Modeling Framework* (EMF) *Ecore* meta-metamodel. In GeKo [12] the composition of the aspect is specified thank to mappings between the pointcut and the base (automatically obtained) and the pointcut and the advice (specified by the designer). All these approaches could be extended with the introduction policies presented in this paper.

In [6], Grønmo *et al.* define a collection operator for graph transformations to ease model transformation. More specifically, the collection operator allows the matching of a set of "similar" join points (subgraphs in graph transformation). In this way, the transformation or the weaving can be applied once on the set of identified join points instead of applying the weaving at the level of each join point. Consequently, this technique can be seen as an example of implementation of our *Global* introduction policy. However, this technique is not flexible enough to specify complex composition strategies. For instance, it is not possible to mix *PerPointcutMatch*, *perRoleMatch*, and *Global* strategies in a same weaving. Since the collection operator is a flexible and interesting solution focusing on join point detection, this technique could be used to complement our approach.

At a programming level, let us consider AspectJ [8]: it is interesting to note that by default an aspect has exactly one instance that cuts across the entire program. Consequently, because the instance of the aspect exists at all join points in the running of a program (once its class is loaded), its advice is run at all such join points. However, AspectJ also proposes some elaborate aspect instantiation directives[11]:

– *Per-object aspects*

[11] More explanation can be found at http://www.eclipse.org/aspectj/doc/next/progguide/

    - *perthis*: If an aspect A is defined *perthis*(*Pointcut*), then one instance of A is created for every object that is the executing object (i.e., "this") at any of the join points matched by Pointcut.
    - *pertarget*: Similarly, if an aspect A is defined *pertarget*(*Pointcut*), then one instance of A is created for every object that is the target object of the join points matched by Pointcut.
- *Per-control-flow aspects*
    - *percflow or percflowbelow*: If an aspect A is defined $percflow(Pointcut)$ or $percflowbelow(Pointcut)$, then an instance of A is created for each flow of control of the join points matched by Pointcut.

7 Conclusions and Perspectives

In this paper, we presented the concept of sharing of model elements introduced by an advice model, which allows designers to specify how the advice should be integrated in the case there are multiple pointcut matches. We defined 4 introduction policies for introduced model elements. By default, new instances of the advice model elements are introduced for each pointcut match (*PerPointcut-Match*). However, it is possible to use other strategies to reuse the same instances for all the pointcut matches (*Global*), or for a given matched set or tuple of model elements in the base model (*PerElementMatch* or *PerRoleMatch*).

We implemented the four policies in the SmartAdapters [14] approach. The paper gives sufficient implementation details so that it could be integrated into any other AOM approach. SmartAdapters is currently used by the industrial partners of the DiVA project to realize real-life case studies (an airport crisis management system and a next-generation customer relationship management system). In DiVA, we are interested in weaving architectural aspects (component, ports, bindings, etc). Even if architectural models are a rather simple domain for AOM, it appears that the notion of sharing of introduced model elements was very useful for our industrial partners. Experience showed that while it is possible to achieve any desired model modification using only *PerPointcutMatch* introduction aspects, it typically requires several aspects with complex pointcut models to achieve the desired effect.

We believe that flexible introduction policies also make writing of reusable aspect models a lot easier. They allow a modeler to state in a simple, base model-independent way under which conditions certain model elements are to be introduced. We intend to conduct experiments to validate this claim in the context of the Reusable Aspect Models (RAM) approach [10] in the near future.

References

1. S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley Professional, April 2005.
2. T. Cottenier, A. van den Berg, and T. Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *AOSD'07: 6th International Conference on Aspect-Oriented Software Development - Industry Track*, Vancouver, Canada, 2007.

3. F. Fleurey, B. Baudry, R. France, and S. Ghosh. A Generic Approach For Automatic Model Composition. In *AOM@MoDELS'07: 11th Int. Workshop on Aspect-Oriented Modeling*, Nashville TN USA, Oct 2007.
4. C. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.
5. I. Groher and M. Voelter. Xweave: Models and aspects in concert. In *AOM Workshop'07 at AOSD*, March 12 2007.
6. R. Grønmo, S. Krogdahl, and B. Møller-Pedersen. A Collection Operator for Graph Transformation. In *ICMT'09: 2nd International Conference on Theory and Practice of Model Transformations*, pages 67–82, Berlin, Heidelberg, 2009. Springer-Verlag.
7. P.K. Jayaraman, J. Whittle, A.M. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, Oct 2007.
8. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. In *ECOOP'01: 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
9. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97: 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, 1997. Springer-Verlag.
10. J. Kienzle, W. Al Abed, and J. Klein. Aspect-Oriented Multi-View Modeling. In *AOSD '09: 8th ACM international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2009. ACM.
11. B. Morin, O. Barais, J. M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *3rd Int. ECOOP'07 Workshop on Models and Aspects, Handling Crosscutting Concerns in MDSD*, Berlin, Germany, August 2007.
12. B. Morin, J.Klein, O. Barais, and J. M. Jézéquel. A Generic Weaver for Supporting Product Lines. In *EA@ICSE'08: Int. Workshop on Early Aspects*, Leipzig, Germany, May 2008.
13. B. Morin, O. Barais, J-M. Jézéquel, F. Fleurey, and A. Solberg. Models@run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.
14. B. Morin, O. Barais, G. Nain, and J-M. Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.
15. P.A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer.
16. C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97: 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, 1997. Springer-Verlag.
17. D. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), February 2006.
18. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *ECOOP'98: 12th European Conference on Object-Oriented Programming*, pages 550–570, London, UK, 1998. Springer-Verlag.
19. J. Whittle, P. Jayaraman, A. Elkhodary, A. Moreira, and J. Araújo. MATA: A unified approach for composing UML aspect models based on graph transformation. *T. Aspect-Oriented Software Development VI*, 6:191–237, 2009.