Model Driven Design and Aspect Weaving

Jean-Marc Jézéquel

SoSyM manuscript No.
(will be inserted by the editor)

Model Driven Design and Aspect Weaving

Jean-Marc Jézéquel[1]

Irisa (INRIA & University of Rennes 1)

Received: 10th December 2007/ Revised version:

Abstract A model is a simplified representation of an aspect of the world for a specific purpose. In complex systems, many aspects are to be handled, from architectural aspects to dynamic behavior, functionalities, user-interface, and extra-functional concerns (such as security, reliability, timeliness, etc.) For software systems, the design process can then be characterized as the weaving of all these aspects into a detailed design model. Model Driven Design aims at automating this weaving process, that is automatically deriving software systems from theirs models. This paper explores the relationship between modeling and aspect weaving. It points out some of the challenges related to such automatic model weaving, illustrating them with the example of a weaving process for behavioral models represented as scenarios[1].

1 Introduction

It is seldom the case nowadays that we can deliver software systems with the assumption that one-size-fits-all [3]. We have to handle many variants accounting not only for differences in product functionalities (range of products to be marketed at different prices), but also for differences in hardware (e.g.; graphic cards, display capacities, input devices), operating systems, localization, user preferences for GUI ("skins"). Obviously, we do not want to develop from scratch and independently all of the variants the marketing department wants. Furthermore, all of these variant may have many successive versions, leading to a two-dimensional vision of product-lines [27].

The traditional way scientists use to master complexity is to resort to modeling. Models can be used for instance to describe and analyze the commonalities and variation points within a software product-line. In the software community however, a lot of misunderstanding on Model Driven Engineering stems from a biased understanding of the nature of modeling.

In this paper, we explore the relationship between modeling and aspect weaving. In Section 2 we recall that a model is indeed a simplified representation of an aspect of the world for a specific purpose. Complex systems typically give rise to more than one model because many aspects are to be handled. For software systems, the design process can be characterized as a (partially automated) weaving of these aspects into a detailed design model. Section 3 then discusses some of the challenges related to such automatic model weaving, concentrating on the need for weaving processes that exhibit good composition properties to allow multiple aspect weavings. As an illustration, it present an example of such a weaving process for behavioral models represented as scenarios. Section 4 presents an implementation environment for building such model weavers, based on the kernel meta-modeling tool Kermeta. Some concluding remarks close the paper.

2 Modeling and Weaving

2.1 Models and Aspects

Modeling is not just about expressing a solution at a higher abstraction level than code. This limited view on modeling has been useful in the past (assembly languages abstracting away from machine code, 3GL abstracting over assembly languages, etc.) and it is still useful today to get e.g.; a holistic view on a large C++ program. But modeling goes well beyond that.

Modeling is indeed one of the touchstone of any scientific activity (along with validating models with respect to experiments carried out in the real world). Note by the way that the specificity of engineering is that engineers build models of artefacts that usually do not exist yet (with the ultimate goal of building them, see Figure 1).

[1] This work has been partially supported by the AOSD-Europe Network of Excellence.

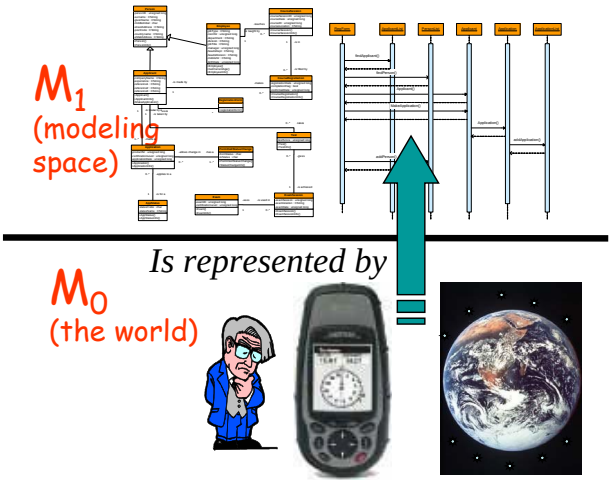


Figure 1 Modeling the world

In engineering, one wants to break down a complex system into as many models as needed in order to address all the relevant concerns in such a way that they become understandable enough [1]. These models may be expressed with a general purpose modeling language such as the UML [26], or with Domain Specific Languages when it is more appropriate (see Figure 2). Each of these models can be seen as the abstraction of an aspect of reality for handling a given concern. The provision of effective means for handling such concerns makes it possible to establish critical trade-offs early on in the software life cycle, and to effectively manage variation points in the case of product-lines.

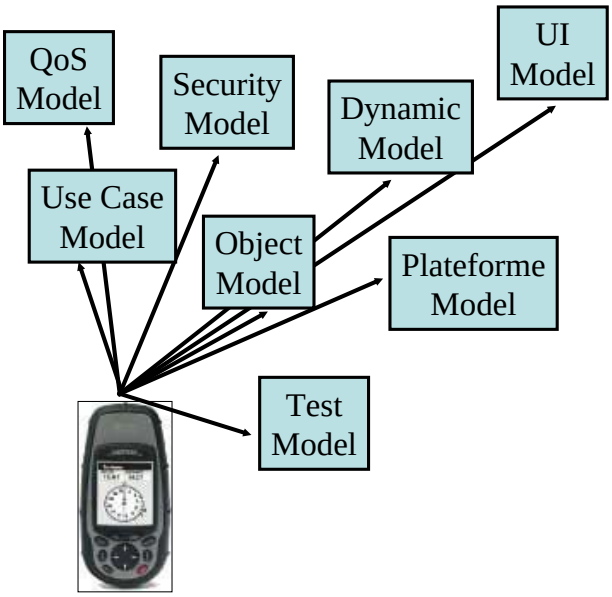


Figure 2 Modeling several aspects

Note that in the Aspect Oriented Programming community, the notion of aspect is defined in a slightly more restricted way as the modularization of a cross-cutting concern [7]. If we indeed have an already existing "main" decomposition paradigm (such as object orientation), there are many classes of concerns for which clear allocation into modules is not possible (hence the name "cross-cutting"). Examples include both allocating responsibility for providing certain kinds of functionality (such as logging) in a cohesive, loosely coupled fashion, as well as handling many non-functional requirements that are inherently cross-cutting e.g.; security, mobility, availability, distribution, resource management and real-time constraints.

However now that aspects become also popular outside of the programming world [24], there is a growing acceptance for a wider definition where an aspect is a concern that can be modularized. The motivation of these efforts is the systematic identification, modularization, representation, and composition of these concerns, with the ultimate goal of improving our ability to reason about the problem domain and the corresponding solution, reducing the size of software model and application code, development costs and maintenance time.

2.2 Design and Aspect Weaving

So really modeling is the activity of separating concerns in the problem domain, an activity that we can call *analysis*. If solutions to these concerns can be described as aspects, the design process can then be characterized as a weaving of these aspects into a detailed design model [11] (also called the solution space, see Figure 3). This is not new: this is actually what designers have been actually doing forever. Most often however, the various aspects are not *explicit*, or when there are, it is in the form of informal descriptions. So the task of the designer is to do the weaving in her head more or less at once, and then produce the resulting detailed design as a big tangled program (even if one decomposition paradigm, such as functional or object-oriented, is used). While it works pretty well for small problems, it can become a major headache for bigger ones.

Note that the real challenge here is not on how to design the system to take a particular aspect into account: there is a huge design know-how in industry for that, often captured in the form of design patterns. Taking into account more than one aspect as the same time is a little bit more tricky, but many large scale successful projects in industry are there to show us that engineers do ultimately manage to sort it out (most of the time).

The real challenge in a product-line context is that the engineer wants to be able to change her mind on which version of which variant of any particular aspect she wants in the system. And she wants to do it cheaply, quickly and safely. For that, redoing by hand the tedious weaving of every aspect is not an option.

We do not propose to solve this problem up-front, but just to mechanize and reproduce the process experienced

designers follow by hand [9]. The idea is that when a new product has to be derived from the product-line, we can automatically replay most this design process, just changing a few things here and there [17].

Usually in science, a model has a different nature that the thing it models (think of a bridge drawing vs. a concrete bridge). Only in software and in linguistics a model has the same nature as the thing it models. In software at least, this opens the possibility to automatically derive software from its model, that is to automate this weaving process. This requires that models are no longer informal, and that the weaving process is itself described as a program (which is as a matter of facts an executable meta-model [19]) manipulating these models to produce a detailed design that can ultimately be transformed to code or at least test suites [21].

This is really what Model Driven Design is all about.

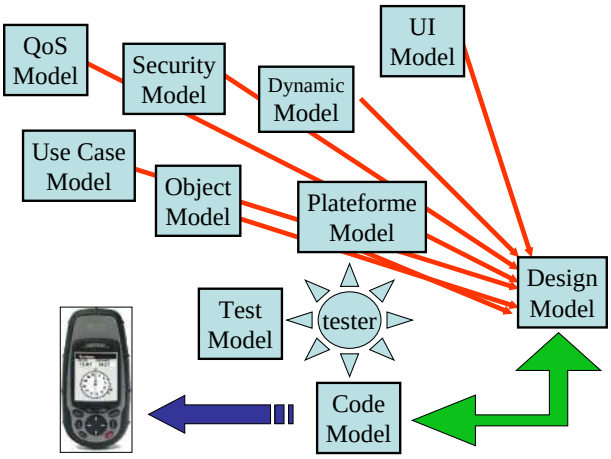


Figure 3 Design is Weaving Models

3 Aspect Weaving in Practice

3.1 Weaving Aspects

Ideally, all aspects are equally important, and should play a symmetrical role. In practice however, a base model is useful to provide a backbone on which other aspects are woven (see Figure 4).

An aspect is then made of:

- A pointcut which is a predicate over a model that is used to select relevant model elements called *join points*.
- An advice which is a new behavior meant to replace (or complement) the matched ones.

Weaving an aspect consist in (1) identifying the join points matching the aspect pointcut and (2) replacing them (or composing them) with the aspect advice.

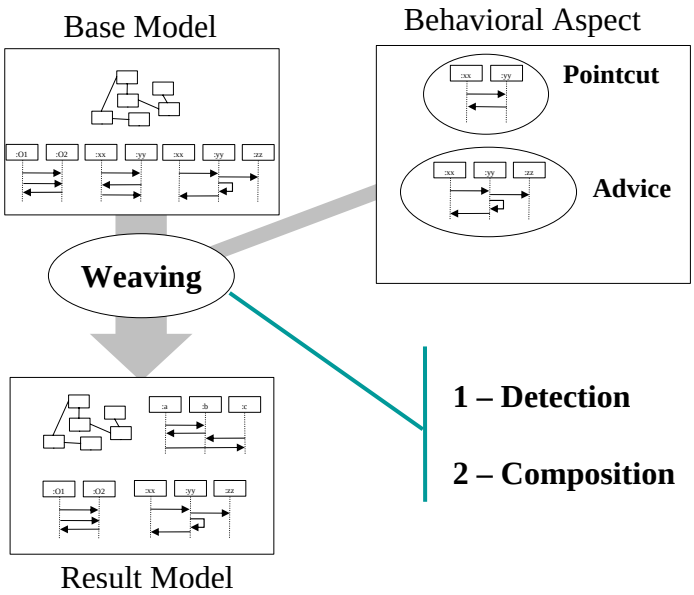


Figure 4 Principle of Aspect Weaving

Both activities of join point identification and model composition can be more complex than it seems though.

First, many complex aspects involve dynamic behavior. This is usually a problem for AspectJ kinds of languages [12] which are limited by their join point models and pointcut expression mechanisms based on concrete syntax [4,16]. With models however, interrelations among model elements (not just classes and objects, but also methods call and other events) can be readily available and identifiable through e.g.; dynamic diagrams. While Class and Object diagrams describe clientship and inheritance among the program elements, Use Cases, Statecharts, Activity and Sequence Diagrams describe how and when the clientship takes place. Therefore, through a simple analysis of the models we can get a much more direct outline of the system execution. In [23] we proposed a framework for expressing aspect pointcuts as model-snippets. The idea is that model-snippets are specified upon concepts in a given domain (meta-model), but that the pattern-matching needed to find join points can be performed generically with respect to meta-models, using a Prolog-like unification. Still, as discussed in [4], it is in general difficult (or even statically undecidable) to identify join points when the patterns we are looking for are based on the properties of the computational flow. For instance consider the following code snippet:

```
void f() {
  while(c()) {
     b(); a();
  }
}
```

Detecting a join point where a call to $a()$ is followed by a call to $b()$ might or might not be statically possible,

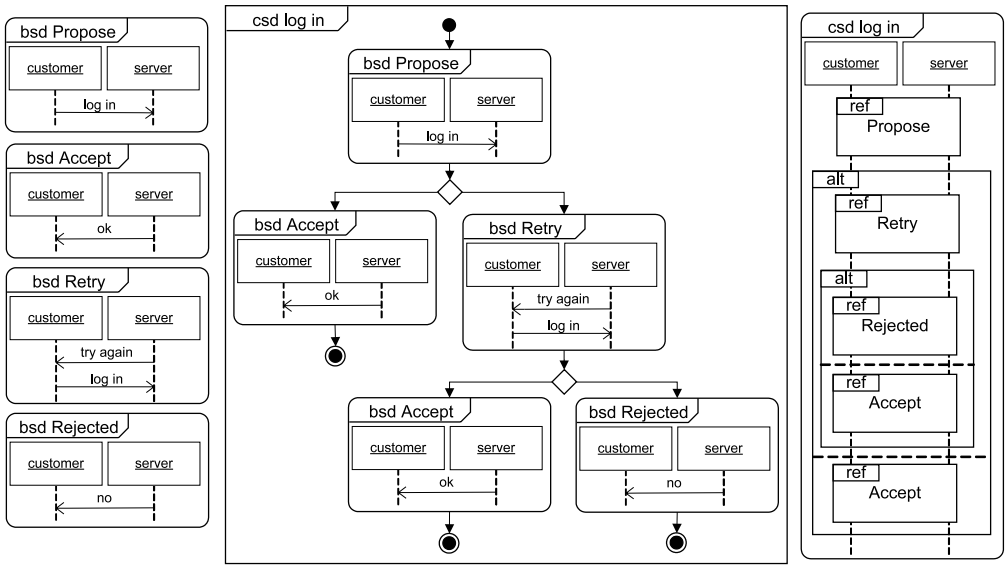


Figure 5 Examples of bSDs and combined SD

depending on the analysis of $c()$. In that particular case, we can see that it would at least require some sort of loop unrolling.

The second dimension of complexity is related to model composition. While weaving a single aspect is pretty straightforward, weaving a second one at the same join point is another story. When a second aspect indeed has to be woven, the join point might not any longer exist because it could have been modified by the first aspect advice. If we want to allow aspect weaving on a pairwise basis, we must then define the join point matching mechanism in a way that takes into account these composability issues. However, with this new way of specifying join points, the composition of the advice with the detected part cannot any longer be just a replacement of the detected part by the advice: we also have to define more sophisticated compositions operators.

The rest of this section investigates these issues taking the example of the simple modeling language of scenarios available in UML2.0 under the form of Sequence Diagrams. Note that even with such a model-level formalisms that directly models possible execution flows, restrictive hypothesis are needed on the input language to make it possible to statically identify the join points [15].

3.2 Weaving Aspects in Sequence Diagrams

Sequence Diagrams are either basic Sequence Diagrams (bSDs), describing a finite number of interactions between objects of the system, or combined sequence diagrams (cSD), that are higher level of specifications that allow the composition of bSDs with operators such as sequence, alternative and loop. Formally, Sequence Diagrams in UML2 are partially ordered sets of event instances. Figure 5 (left) shows several bSDs which describe some interactions between the two objects *customer* and *server*. The vertical lines represent life-lines for the given objects. Interactions between objects are shown as arrows called messages like *log in* and *try again*. Each message is defined by two events: message emission and message reception which induces an ordering between emission and reception.

These bSDs can be composed with operators such as sequence, alternative and loop to produce a more complex Sequence Diagram (also called UML 2.0 Interaction Overview Diagram). In Figure 5 (center), the cSD *log in* represents a customer log in on a server. The customer tries to log in and either succeeds, or fails. In this last case, the customer can try again to log in, and either succeeds, or the server answers "no". Figure 5 (right) shows a more compact view of the same cSD, allowing an alternative between the bSDs Accept and Retry, and between the bSDs Accept and Rejected.

In this context, we define a *behavioral aspect* as a pair $A = (P, Ad)$ of bSDs. P is a pointcut, i.e. a bSD interpreted as a predicate over the semantics of a base model satisfied by all join points. Ad is an advice, i.e. the new behavior that should replace the base behavior when it is matched by P.

When we define aspects with sequence diagrams, we keep some advantages related to sequence diagrams. In particular, it is easy to express a pointcut as a sequence of messages. Figure 6 shows three behavioral aspects. The first one allows the persistence of exchanges between the customer and the server. In the definition of the pointcut, we use regular expressions to easily express three kinds of exchanges that we want to save (the message *log in* followed by either the message *ok*, the message *try again*, or the message *no*). The second aspect allows the identification of a login that fails. The third aspect allows the addition of a display and its update.

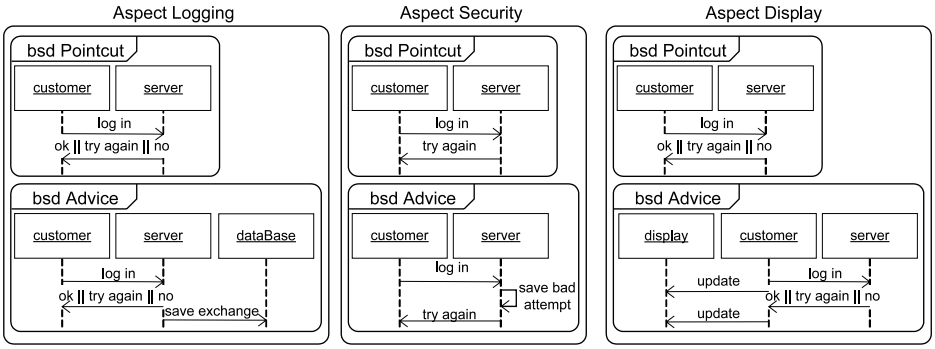


Figure 6 Three behavioral aspects

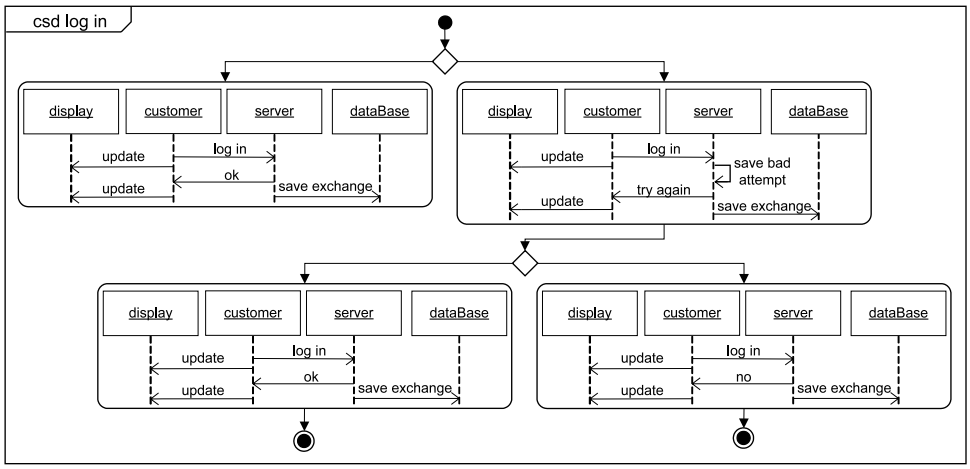


Figure 7 Result of the weaving

The expected weaving of the three aspects depicted in Figure 6 into the cSD *log in* is represented by the cSD in Figure 7.

3.3 Weaving More Than One Aspect: the Detection Problem

Weaving several aspects at the same join point can be difficult if a join point is simply defined as a strict sequence of messages, because aspects previously woven might have inserted messages in between. In this case, the only way to support multiple static weaving is to define each aspect in function of the other aspects, which is clearly not acceptable.

The weaving of the three aspects depicted in Figure 6 allows us to better explain the problem. If the join points are defined as the strict sequence of messages corresponding to those specified in the pointcut, the weaving of these three aspects is impossible. Indeed, when the aspect *security* is woven, a message *save bad attempt* is added between the two messages *log in* and *try again*. Since the pointcut only detects a strict sequence of messages, after the weaving of the aspect *security*, the aspect *display* cannot be woven anymore. We obtain the same problem if we weave the aspect *display* first and the aspect *security* afterwards.

To solve this problem of multiple weaving, we need definitions of join points which allow the detection of join points where some events can occur between the events specified in the pointcut. In this way, when the aspect *security* is woven, the pointcut of the aspect *display* will allow the detection of the join point formed by the messages *log in* and *try again*, even if the message *save bad attempt* has been added.

In our approach, the definition of join point will rely on a notion of *part of a bSD*, which is a subset of a bSD where any kind of messages can occur between the messages of the pointcut. A join point will then be defined as a part of the base bSD such that this part corresponds to the pointcut.

The notion of correspondence between a part and a pointcut is defined as an isomorphism between bSD, made of a set of 3 isomorphisms between the base SD and the pointcut SD:

f_0 is an isomorphism for matching objects;
f_1 is an isomorphism for matching events;
f_2 is an isomorphism for matching message names (taking into account wild-cards).

As an example, figure 8 shows a bSD morphism $f =< f_0, f_1, f_2 >: pointcut \rightarrow M2$ where only the morphism f_1 associating the events is represented (for instance, the

event ep_1 which represents the sending of the message $m1$ is associated with the event em_2).

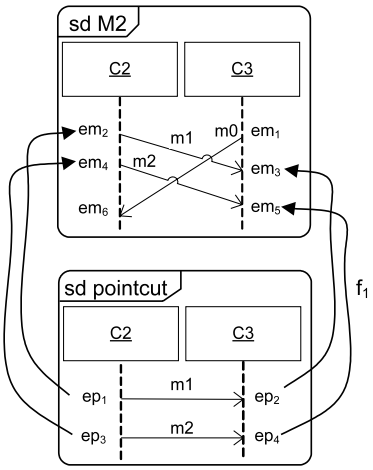


Figure 8 Example of a morphism between bSD

3.4 Weaving More Than One Aspect: the Composition Problem

Detecting a join point in a bSD then boils down to construct such an isomorphism between a pointcut and a base bSD. Once such a join point has been detected, it remains to compose the bSD Advice with the join points. Since some messages can be present between the messages forming the join point, it is not possible to simply replace a join point by an advice because we would lose the "in-between" messages. Therefore, we have to define a new operator of composition which takes into account the common parts between a join point and an advice to produce a new bSD which does not contain copies of similar elements of the two operands [14]. We use an operator of composition for bSDs called *left amalgamated sum*, inspired by the amalgamated sum proposed in [13]. We add the term $left$ because our operator is not commutative, as it imposes a different role on each operand.

Figure 9 shows an example of left amalgamated sum where the two bSDs *base* and *advice* are amalgamated. For that, we use a third bSD which is the *pointcut* and two bSD morphisms f : *pointcut* $\rightarrow$ *base* and g : *pointcut* $\rightarrow$ *advice* which allow the specification of the common parts of the two bSDs *base* and *advice*.

f is the isomorphism from the *pointcut* to M' that has automatically been obtained with the process of detection described into the previous section.

The morphism g, which indicates the elements shared by the advice and the pointcut, has to be specified when the aspect is defined. In this way, g allows the specification of abstract or generic advices which are "instantiated" by the morphism. For instance, it is not mandatory that the advice contains objects having the same names as those present in the pointcut. In the three aspects in Figure 6, the morphism g is not specified but it is trivial: for each aspect, we associate the objects and the actions having the same names, and the events corresponding to the actions having the same name. The advice of the aspect Display in Figure 6 could be replaced by the "generic" Advice in Figure 9. It is the morphism g which indicates that the object *customer* plays the role of the object *subject* and that the object *server* plays the role of the object A.

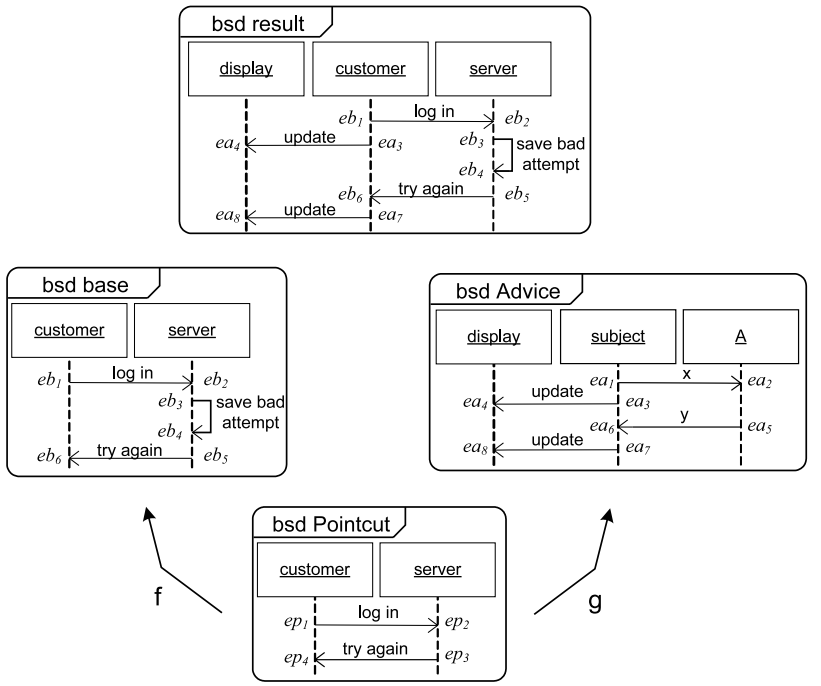
In Figure 9, the elements of the bSDs *base* and *advice* having the same antecedent by f and g will be considered as identical in the bSD *result*, but they will keep the names specified in the bSD *base*. For instance, the objects *subject* and A in the bSD *advice* are replaced by the objects *customer* and *server*. All the elements of the bSD *base* having an antecedent γ by f such that γ has not an image by g in the bSD *advice* are deleted. This case does not appear in the example proposed, but in this way we can delete messages of the bSD *base*. For instance, in an amalgamated sum, if the right operand (the bSD advice in the example) is an empty bSD then the part of the left operand which is isomorphic to the *pointcut* (that is to say the join point), is deleted. Finally, all the elements of the bSDs *base* and *advice* having no antecedent by f and g are kept in the bSD *result*, but the events of the bSD *advice* will always form a "block" around which the events of the bSD *base* will be added. For instance, in Figure 9, in the bSD *base*, if there were an event e on the object *customer* just after the message *try again*, then this event e would be localized just after the sending of the message *update* (event ea_7) in the woven SD.

Note that if we are interested in carrying on an implementation based on these woven SD, we could easily synthesize statecharts from them [25], and then either derive test cases [21] or an implementation [5].

3.5 Discussion

This example with Sequence Diagrams illustrates several hard issues of weaving aspects at model level, involving both semantic based pattern matching and semantic based composition. While our solutions are very specific to the semantics of Sequence Diagrams, we can find related problems and solutions in other modeling languages, for instance Statecharts or Activity Diagrams.

It probably means that there is no hope for a fully general purpose, meta-model independent, model-level aspect weaver. Still, it should be possible to develop aspect weaving software components handling several aspects of aspect weaving, from general purpose model-level pattern matching [23] to automated support for composing models written in a particular language (through a definition of model composition behavior in the metamodel defining the language [8]), to specializ-



f=(f₀,f₁,f₂): Pointcut → base, where
- *f₀: $I_P \rightarrow I_b$ is the identity*
- *f₁ respectively sends ep₁, ep₂, ep₃ and ep₄ to eb₁, eb₂, eb₅ and eb₆*
- *f₂: $A_p \rightarrow A_b' \subset A_b$ is the identity*

g=(g₀,g₁,g₂): Pointcut → advice, where
- *g₀: $I_P \rightarrow I_a' \subset I_a$ sends customer to subject and server to A*
- *g₁ respectively sends ep₁, ep₂, ep₃ and ep₄ to ea₁, ea₂, ea₅ and ea₆*
- *g₂: $A_p \rightarrow A_a' \subset A_a$ sends log in to x and try again to y*

Figure 9 An example of left amalgamated sum

able model composers [10]. These aspect weaving components could then be customized and combined to build domain specific or even project specific aspect weavers.

From an higher viewpoint, what we are trying to achieve is to reify the design process into a weaver program that makes it possible to re-build as often as it is need the target software from its models. The goal is to have only small modifications to make when requirements do change. Actually we need a family of weavers, that for a given product-line are just variants of one's another [17].

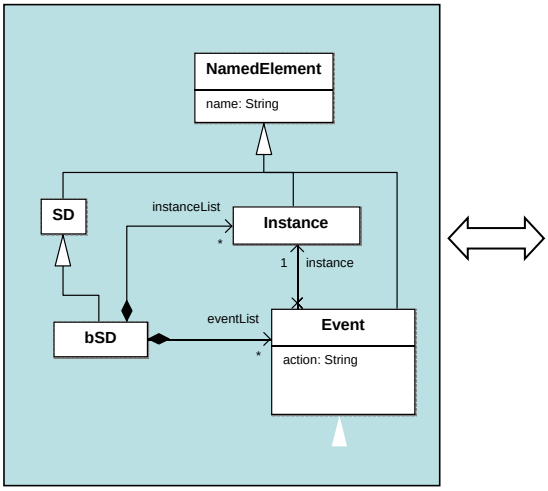
There is thus a need for tools to build these weavers efficiently. These tools are often called meta-tools, or more properly meta-modeling tools, that is tools to build tools to build software from models. Several of these meta-modeling tools have matured over the last decade, the most well known being Metacase [22], Xactium [6], GME [18], and Kermeta [19]. All these meta-modeling tools have in common that they are based on an executable meta-modeling language specifically designed to support the design of tools dedicated to user defined meta-models. In the next section, we outline one of these tools, Kermeta, and describe how it can be used to build a model weaver for our Sequence Diagrams example.

4 Building Project Specific Aspect Weavers with Kermeta

Kermeta is a meta-modeling language which allows describing both the structure and the behavior of meta-models. It has been designed to be compliant with the OMG meta-modeling language EMOF (part of the MOF 2.0 specification) and Ecore (from Eclipse). It provides an action language for specifying the behavior of models. Kermeta is intended to be used as the core language of a model oriented platform. It can be seen as a common basis to implement Metadata languages, action languages, constraint languages or transformation language. Kermeta is statically typed, with generics as well as function types to allow OCL style forall/exist/iterate style of closures. It also directly supports model-oriented concepts like associations, multiplicities or object containment management, as well as aspect-oriented style of static introduction[2].

A MOF meta-model is a valid Kermeta program that just declares packages and classes and does nothing (see Figure 10). Several aspects can then be introduced incrementally to *breath life* into this meta-model, either

[2] Since a Kermeta program is a model, it can also obviously reflectively support the kind of aspect weaving described in this article.




```
package bigSd;
using kermeta::standard
using kermeta::persistence
abstract class NamedElement
{
    attribute name : String[1..1]
}
abstract class SD inherits NamedElement
{}
class BSD inherits SD
{
    attribute eventList : Event[0..*]
    reference instanceList : Instance[0..*]
    ...
}
abstract class Event inherits NamedElement
{
    attribute action : String[1..1]
    reference instance : Instance[1..1]
    ...
}
....
```

Figure 10 Extract of the bSD Metamodel and its corresponding Kermeta textual syntax

for handling concerns of static semantics, dynamic semantics, or model transformations [20] . Here, we are interested in the later, i.e. adding Kermeta code that weave several behavioral aspects into a base scenario.

As illustrated in Figure 11, the weaving implemented with Kermeta consists of two steps. First, the detection step uses the pointcut model and the base model to compute a set of join points. Each join point is characterized by a morphism from the pointcut to a corresponding elements in the base model. Secondly, using these morphisms, the advice is composed with each join point in the base model.

The first step processes models to extract join points and the second is a model transformation. Figure 12 gives an other view of the overall process, concentrating on the input and output models of these two steps (each ellipse is a model and the black rectangle on the top left-hand corner indicates its meta-model). Except for morphisms (which are defined with their own metamodels), all models are SDs.

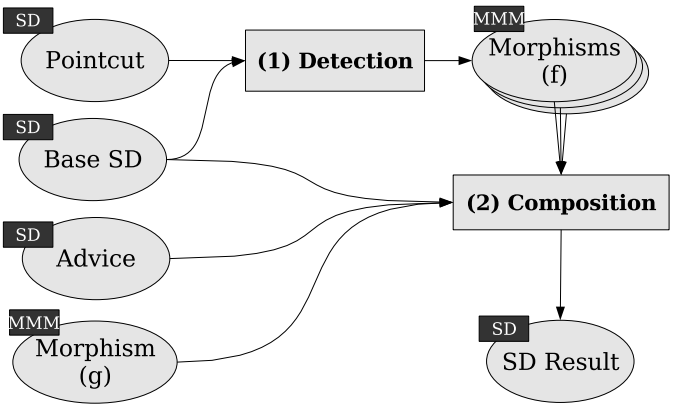


Figure 12 Transformation of Models

5 Conclusion

From an engineering point of view, modeling and weaving aspects are symmetric activities: weaving aspects is the process by which analysis models are transformed into a design model. Model Driven Engineering makes it possible to automate this process: i.e. to have software build software instead of building it by hand. In this paper we have described the overall vision of a design process based on these ideas.

We have highlighted however that both activities of join point identification and model composition can be more complex than it looks at first. Beyond the well known fact that it is in general difficult (or even statically undecidable) to identify join points when the patterns we are looking for are based on the properties of the computational flow, additional difficult join point detection and composition problems may arise when more than one aspect has to be woven. When a second aspect indeed has to be woven, the join point might not any longer exist because it could have been modified by the first aspect advice. For aspect weaving to be defined on a pair-wise basis, the join point matching mechanism must take into account these composability issues. Further, with this way of specifying join points, the composition of the advice with the detected part cannot any longer be just a replacement of the detected part by the advice: we also have to define more sophisticated compositions operators.

We have illustrated these issues on a toy example where simple aspects, described as Sequence Diagram pairs (representing the aspect pointcut and advice), had to be woven into a base Sequence Diagram.

While it probably means that there is no hope for a fully general purpose model-level aspect weaver, we have shown that it is possible to develop aspect weav-

```
require kermeta   require "../models/bigSd.kmt" require "../detectionAlgorithm/Detection.kmt"
require "../amalgamatedSum/LeftSum.kmt"
using kermeta::standard    using bigSd

class Weaver {
 operation weave(base : BSD, pointcut : BSD, advice : BSD, g : BSDMorphism) : BSD is do
      result := BSD.new                                                          Initialization
      //Declaration of the various components we need
      var detection: Detection.new
      var sum: LeftSum init LeftSum.new
      var f: BSDMorphism init BSDMorphism.new
      var setOfMorphism : Set< BSDMorphism > init Set< BSDMorphism >.new

      //Detection Step                                                           Detection Step
      f:= detection.detect(pointcut, base)
      while (f != null)
           setOfMorphism.add(f)
           f:= detection.detect(pointcut, minus(base,f))
      end

                                                                               Composition Step
      //Composition Step
      setOfMorphism.each{f | result := sum.merge(result, pointcut, advice, f, g)
end
```

Figure 11 Weaving Aspects in Kermeta

ing software components handling several aspects of aspect weaving, from general purpose model-level pattern matching to ad hoc and specializable model composers. These aspect weaving components could then be customized and combined to build domain specific or even project specific aspect weavers using for instance a an executable meta-modeling environment as available in Kermeta.

Still there are additional important issues that we did not specifically address in this article, for instance the problem of weaving heterogeneous models [2], that is models consisting of different diagram types as available in UML. In that case, we have to be concerned with consistency of the models: does weaving take place only when information is represented consistently across the diagrams or should weaving tolerate the inconsistencies and clearly bring out problems arising from inconsistencies during or after the weaving? Answering these questions, as well as others including process issues such as how interactive vs. automatic should be the weaving, or when should the weaving take place (i.e. at once at modeling time, or in several steps including some steps carried out at runtime) will be the subject of much future works in our research team.

Acknowledgments

This work has been partially supported by the AOSD-Europe Network of Excellence.

We are also extremely grateful to Jacques Klein and Franck Fleurey for the development of the weaving example used all along this paper, and beyond that, for many interesting discussions around aspect weaving.

References

1. Elisa Baniassad and Siobhán Clarke. Theme: An approach for aspect-oriented analysis and design. *icse*, 0:158–167, 2004.
2. Olivier Barais, Jacques Klein, Benoit Baudry, Andrew Jackson, and Siobhan Clarke. Composing multi-view aspect models. In *7th IEEE International Conference on Composition-Based Software Systems (ICCBSS)*, Madrid, Spain, February 2008.
3. J. Bosch. Software product families in Nokia. In *Proc. 9th Int. Conference on Software Product Lines*, number 3714 in LNCS, pages 2–6, Rennes, France, September 2005. Springer.
4. Walter Cazzola, Jean-Marc Jézéquel, and Awais Rashid. Semantic join point models: Motivations, notions and requirements. In *SPLAT 2006 (Software Engineering Properties of Languages and Aspect Technologies)*, March 2006.
5. Franck Chauvel and Jean-Marc Jézéquel. Code generation from UML models with semantic variation points. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages –, Montego Bay, Jamaica, October 2005. Springer.
6. T. Clark, A. Evans, P. Sammut, and J. Willans. Applied Metamodelling: A Foundation for Language Driven Development. *Available for download from www. xactium. com*, 2004.
7. R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000.
8. Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In *EDOC'07 (Entreprise Distributed Object Computing Conference)*, Annapolis, MD, USA, 2007.
9. Wai Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h, and Noël Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of 1st ACM*

International Conference on Aspect Oriented Software Development, AOSD 2002, Enschede, The Netherlands, April 2002.
10. Andrew Jackson, Olivier Barais, Jean-Marc Jézéquel, and Siobhán Clarke. Toward a generic and extensible merge operator. In *Models and Aspects workshop, at ECOOP 2006*, Nantes, France, July 2006.
11. Jean-Marc Jézéquel, Noël Plouzeau, Torben Weis, , and Kurt Geihs. From contracts to aspects in uml designs. In *Proc. of the Workshop on Aspect-Oriented Modeling with UML at AOSD'02*, 2002.
12. G. Kiczales. The fun has just begun. *Keynote address at AOSD. Available at aosd. net/archive/2003/kiczales-aosd-2003. ppt*, 2003.
13. Jacques Klein, Benoit Caillaud, and Loic Hélouët. Merging scenarios. In *9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, pages 209–226, Linz, Austria, sep 2004.
14. Jacques Klein, Franck Fleurey, and Jean-Marc Jézéquel. Weaving multiple aspects in sequence diagrams. *Trans. on Aspect Oriented Software Development*, pages 85–96, December 2007.
15. Jacques Klein, Loic Hélouet, and Jean-Marc Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, March 2006. ACM.
16. Jacques Klein and Jean-Marc Jézéquel. Problems of the semantic-based weaving of scenarios. In *In Aspects and Software Product Lines: An Early Aspects Workshop at SPLC-Europe 05*, Rennes, September 2005.
17. Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. Introducing variability into aspect-oriented modeling approaches. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, USA, October 2007.
18. A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. *Workshop on Intelligent Signal Processing, Budapest, Hungary, May*, 17, 2001.
19. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.
20. Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, October 2005.
21. Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel, and Yves Le Traon. Test synthesis from UML models of distributed software. *IEEE Transactions on Software Engineering*, 33(4):252–268, April 2007.
22. R. Pohjonen and J.P. Tolvanen. Automated Production of Family Members: Lessons Learned. *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Workshop on Product Line Engineering*, 2002.
23. Rodrigo Ramos, Olivier Barais, and Jean-Marc Jézéquel. Matching model-snippets. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, USA, October 2007.
24. A. Rashid and J. Araújo. Modularisation and composition of aspectual requirements. *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 11–20, 2003.
25. Tewfic Ziadi, Loic Hélouët, and Jean-Marc Jézéquel. Revisiting statechart synthesis with an algebraic approach. In *26th International Conference on Software Engineering (ICSE 04)*, ACM, pages 242–251, Edinburgh, UK, May 2004.
26. Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a UML profile for software product lines. In *Proceedings of the Fifth International Workshop on Product Familly Engineering (PFE-5)*, volume 3014 of *LNCS*, pages 129–139. Springer Verlag, 2003.
27. Tewfik Ziadi and Jean-Marc Jézéquel. *Software Product Lines*, chapter Product Line Engineering with the UML: Deriving Products, pages 557–586. Number ISBN: 978-3-540-33252-7. Springer Verlag, 2006.