Journal of Computing and Information Technology - CIT 14, 2006, 2, 111–131
doi:10.2498/cit.2006.02.03

From Composition Filters to AspectJ: A Platform Specific Model Transformation

Djamel Meslati[1], Mohamed T. Kimour[1] and Saïd Ghoul[2]

[1] Laboratory of Research on Computer Science (LRI), University of Annaba, Algeria
[2] Computer Science Department, Philadelphia University, Amman, Jordan

Both model-driven architecture (MDA) and aspect-oriented programming (AOP) are promising paradigms that are very attractive for the software engineering community. While the former is an approach to application design and implementation using models as first class entities, the latter advocates the separation of concerns as an approach to tackle most software development and maintenance problems. MDA and AOP can be related in various ways and their combination seems to be a promising issue. In this article, we focus on the transformation of two AOP approaches, composition filters (CF) and ASPECTJ, considered as platform specific metamodels within the MDA context. We propose a transformation of CF models into ASPECTJ models using a syntax-directed translation and a set of transformation templates. In addition to being easy to implement, our transformation approach covers the most important concepts of CF.

Keywords: aspect-oriented programming, ASPECTJ, composition filters, model-driven architecture, model transformation.

1. Introduction

Model-driven architecture is an approach to system development that provides means for using models to direct the course of systems understanding, design, construction, deployment, operation, maintenance and modification [18]. The core concepts of MDA are models, metamodels and transformations. A model is a representation of a part of the function, structure and/or behavior of a system. A model specification can be formal when it is based on a language that has a well defined syntax, semantics and possibly rules to analyze its constructs [17]. Under this definition, a source code is a model that has the salient characteristic that it can be executed by a machine [7, 16, 17].

A metamodel is a special kind of model that specifies the abstract syntax of a modeling language. In the MDA context, each model is an instance of a metamodel that is described using the OMG MOF (Object Management Group Meta Objet Facility) [19]. A transformation is the process that converts one model to another model of the same system according to some description which, in turn, is a model and an instance of some metamodel [14, 18].

Broadly speaking, MDA is an approach where models are first class entities [8, 14]. A software system can be seen as a collection of models of various abstraction levels where each describes the system from some viewpoint and, consequently, most engineering tasks can be considered as modeling and transforming models.

In order to promote interoperability and portability, the MDA approach puts the emphasis on two kinds of models with respect to specific platforms: the platform independent models (PIM) and the platform specific models (PSM). Within each of these abstractions, there can be a number of refinements (for example, many levels of platform independent models) [14]. For example, a program is a PSM obtained throughout a cascade of models and transformations. At the top, we find a use case PIM that highlights aspects of the system corresponding to the user view. Then, the use case PIM is transformed to extract objects and classes which form another PIM. The last transformation might be a

JAVA model which is specific to a JAVA platform (i. e. the JAVA virtual machine). Notice that a "platform specific" is meaningful only relative to a particular point of view [8]. For example, the JAVA program is a PSM when considering a specific JAVA platform, while it is a PIM when considering the specific operating systems on which the JAVA virtual machine is implemented.

In the MDA approach, transformations can be of various types such as merging or composing models, but usually they convert models offering a particular view from one level of abstraction to a less abstract one (e.g. PIM to PSM or PIM to PIM), by adding more details supplied by the transformation rules [14]. Transformations between models are needed because:

- Metamodels use various modeling concepts and notations to highlight one or more views within a model, depending on what is relevant at any point in time [18].
- Metamodels influence the modeling task and the perception we have of the real world. Thus it is beneficial to use various metamodels to capture the real world subtle situations during the engineering tasks [8].
- According to "divide and conquer" principle, the engineering tasks can be simplified if complex system models are composed or merged from less complex models using transformations that supply necessary details.

In this article, we focus on the transformation of two AOP approaches, CF and ASPECTJ, by considering them as two platform specific metamodels within the MDA context. This work is a part of an ongoing one that aims to construct an MDA environment where software developers can use multiple AOP metamodels, during the engineering tasks, and freely switch from one another, using automated tools that preserve the concerns' traceability.

AOP is a particular separation of concerns approach (SOC), where a system can be seen as composed of business logic and concerns such as synchronization, security, persistence, etc. It is now commonly admitted that an appropriate separation of concerns have an influence on the development and maintenance processes. It reduces the software complexity and code tangling, facilitates reuse, improves comprehensibility, simplifies component integration and decreases invasive changes [2, 12, 20].

Today, a large amount of literature is devoted to three SOC approaches: composition filters (CF) [5], ASPECTJ [12] and hyperspace approach [20]. They aim at providing better concepts and mechanisms to appropriately separate the software concerns from the business logic. Unfortunately, each approach has its own philosophy and concepts. An approach might be suitable from some point of view, but inappropriate from another. Consequently, providing an environment where multiple AOP approaches can be used simultaneously is a worthy goal. This article is an attempt towards this goal that focuses only on the transformation of CF models to ASPECTJ models. Our interest in CF and ASPECTJ results from the fact that they are both AOP approaches.

The remainder of this article is composed of six sections. Sections 2 and 3 describe, respectively CF and ASPECTJ metamodels in an intuitive way. In section 4 we give the motivations of the work and in section 5 we present CF to ASPECTJ transformation. Section 6 discusses related work, and section 7 conclusion and future work.

2. The Composition Filters Metamodel

2.1. Principle and Goals

CF considers a system as a set of objects that interact with each other to achieve a common task. Most interactions are done by sending and receiving messages and CF intervenes during these interactions [3, 6]. It provides an object with an interface containing filters that intercept and manipulate messages in various forms, modifying their scope and expected behavior. The former consists of delegating messages to other objects (i. e. changing the target object to which the message is sent), whereas the latter consists of substituting a message selector with another (i. e. replacing the name of the method to be called with the name of another one to be called instead). By controlling messages (changing their targets and/or selectors) and through a well-constructed interface, CF provides suitable solutions to many problems (see [2]), such as:

- Dynamic inheritance by enabling and disabling inheritance relation between classes at runtime
- Modeling of state dependent behaviors where the behavior of an object changes according to its state
- History sensitivity where the behavior of an object depends on its previous behaviors
- Providing multiple views of the same object (see example in 2.3)
- Behavior coordination and synchronization
- Tracing of programs during the development

One of the CF strengths is the use of a uniform filtration mechanism to resolve the above problems. From this point of view, CF is easy to understand and work with as it only adds few concepts to the object metamodel.

2.2. Basic Concepts

CF adds to an object a wrapping layer called `interface` that traps incoming and outgoing messages. Figure 1 depicts the contents of an interface added to a kernel object. We refer to object or class with a CF interface by CF object (respectively CF class).

A CF interface is composed of the following parts:

- **Internal objects** are objects whose methods are used to compose the behavior of the CF object. Messages received by a CF object can be delegated to internal objects instead of the kernel object. Internal objects are encapsulated in the CF object and cease to exist when the CF object is garbage collected.
- **External object** are almost like internal objects. However, they are supposed to exist on their own and their references are passed on to the CF class constructor during instantiation. These references are assigned to the corresponding CF instance variables.
- **Methods**: Contains all the public methods of the kernel class.
- **Conditions**: Conditions are specific methods without parameters that supply information about the context of a call and the kernel state without changing them [6].
- **Input filters**: A set of declarative specifications that handle the incoming messages.
- **Output filters**: A set of declarative specifications that handle the outgoing messages.

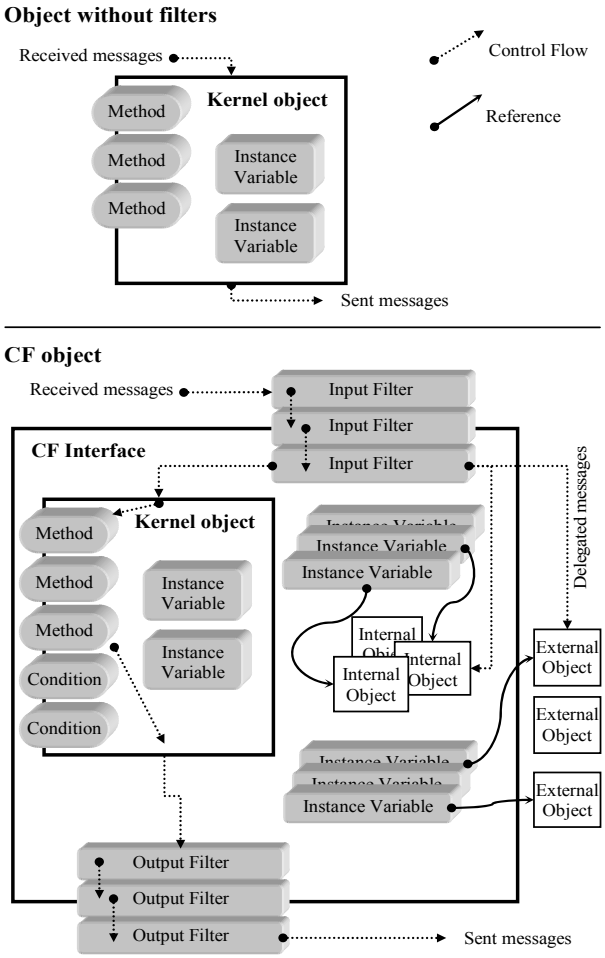


Fig. 1. An object before and after adding a CF interface.

The signature of a CF object is the set of public methods that it responds to. This includes public methods of the kernel class and public methods of the internal/external objects. In the case where two methods have the same name, that of the kernel or the internal/external object declared first in the interface, hides the other. Within a CF object, the kernel and internal/external objects are called targets. The target of a message is determined by the CF object itself when a message is received and becomes a data accessible to filters.

Filters are declared in ordered sets as declarative specifications. A call entering a CF object

is first reified (i. e. the method selector becomes accessible and target determined), then passes each filter in the set until it is discarded or dispatched. Discarding a call raises an exception, whereas dispatching consists of:

- Activating the corresponding method in the kernel or internal/external objects, or
- Substituting it with a call to another method in the kernel or internal/external objects and activating it.

Each filter can accept or reject a call with an effect depending on the semantic of its type. Accept may imply dispatching or simply ignoring the message which, then, passes to the next filter. Reject may imply that a message is discarded, queued as long as the filter expression results in a rejection, or merely ignored (i. e. the message continues with the next filter, (see Table 1). There are five commonly used filter type: `Error`, `Dispatch`, `Wait`, `Meta`, and `RealTime` [5]. Each type deals with a certain category of concerns, but in general a filter set contains more than one type. `Wait` is used to model synchronization concerns, `Meta` allows the reification of a message so that access to its arguments, sender, receiver, return value, becomes possible and `Realtime` deals with timing constraints. `Error` and `Dispatch` are used alone or in combination with other filter types to allow the modeling of various concerns. All filter types can be used in input and output filters, except `Dispatch` which is used only in input filters. In many CF articles, authors consider that output filters operate almost like input filters and do not require specific treatment. Moreover, the object-oriented paradigm tends to adopt a client/server model where the server responsibility is prevalent. Therefore, concerns are usually related to servers rather than scattered among several clients (i. e. modeled in an input filter set rather than in several output ones [4]). For these reasons, output filters are not considered in this article.

To enhance the descriptive power of CF, each filter is composed of several elements called *filter element* (FE) that have the following form:

```
Filtername : Type = {FilterElement,
FilterElement, ...}
```

An incoming message passes through each filter element which accepts or rejects it. Again, reject or accept meaning depends on the filter type (see Table 2). Each filter element specifies a condition `C` and a list of pairs (matching part, substitution part). We call it `MSPList` for short. A FE accepts a call if the condition is true and if the call matches the `MSPList`. Figure 2 depicts the syntactic diagrams corresponding to different forms of filter elements.

To simplify the filter set specification, CF proposes two operators $\Rightarrow$ and $\sim>$ called respectively inclusion and exclusion operators. `C` $\Rightarrow$ `MSPList` means that when the condition `C` is true, any message that matches `MSPList` will be accepted. `C` $\sim>$ `MSPList` means that when the condition `C` is true, all messages, except those in `MSPList`, will be accepted.

In the following, we use two filter examples to illustrate the syntactic diagrams of Figure 2.

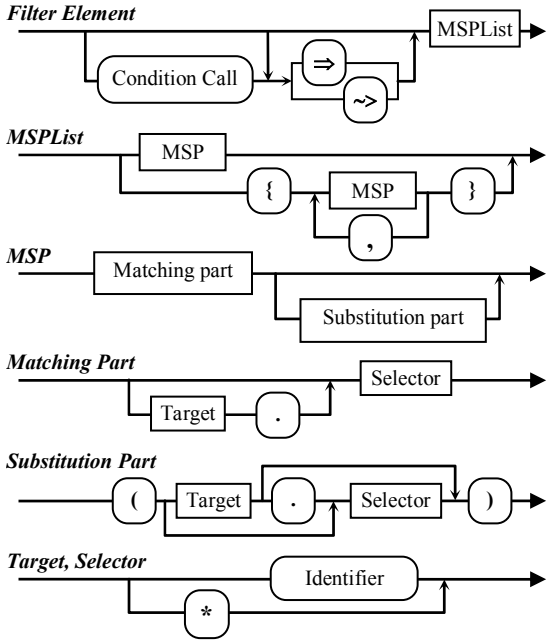


Fig. 2. Syntactic diagrams of filter elements.

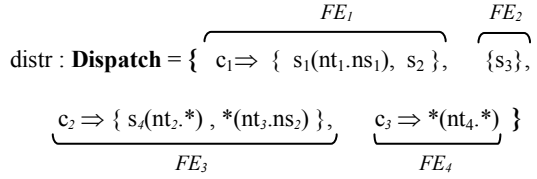
As an example, let `err` be an `Error` filter defined in the CF interface of a class called `Ker`. `FE` stands for filter element, `C` for condition and `S` for message selector.

$$\text{err} : \textbf{Error} = \{ \overbrace{c_1 \Rightarrow \{ s_1, s_2 \}}^{FE_1}, \overbrace{c_2 \sim> \{s_4\}}^{FE_2}, \overbrace{c_3 \sim> *}^{FE_3}, \overbrace{\sim> s_5}^{FE_4} \}$$

FE_1 specifies that when C_1 is true, only calls to S_1 and S_2 are accepted. FE_2 specifies that if C_2

is true, only the call S_4 is rejected. FE_3 has a "*" as a selector to specify that any message is rejected when C_3 is true. FE_4 rejects all calls to S_5.

Now, let us consider a dispatch filter `distr` defined in the CF interface of class `Ker.nt` stands for a new target.



FE_1 is a filter element with two parts. The first one can be written $C_1 \Rightarrow S_1(nt_1.ns_1)$. It has all the needed information (i. e. condition, selector, new target and new selector) and specifies that when C_1 is true, messages having selector S_1 are delegated to method ns_1 of object nt_1, which is an internal or external object. The second part can be written $C_1 \Rightarrow S_2$, it means that messages having selector S_2 are accepted and executed by the kernel object itself. FE_2 lacks a condition and an operator. In this case, default value `True` and inclusion operator are assumed instead. FE_3 has a "*" as message selector. In the beginning "*" means accept any message and instead of the new selector it indicates that selector S_4 will remain unchanged (i. e. the new selector is S_4 itself). FE_4 specifies that when the condition C_3 is true, any call to any methods is delegated to nt_4 using the same selector. nt_1, nt_2, nt_3 and nt_4 must be internal or external objects. Table 1 summarizes the acceptance and rejection meaning for a filter according to its type. Table 2, shows when a filter element accepts or rejects a message and the effect on the filter level (see [4] for more details).

In Table 2, we consider that an incoming message has `T` as the target object and `S` as selector.

According to the value of condition `C`, and if there is a matching with `MSPList`, the effect on the filter element will be to accept/reject a call and, in turn, this will have an effect on the whole filter, which may accept or reject the call or merely let the message continue with the next FE.

2.3. Example

As an illustration, let us consider an Email system example. The system is composed of four classes `Sender`, `Recipient`, `DeliveryAgent`, and the main class called `Email` which is partially shown in listing 1.

Filter Type	Handler actions
`Error`	**Accept**: The accepted message continues with the next filter in the filter set. **Reject**: An exception is raised.
`Meta`	**Accept**: The accepted message is reified as an object of class `Message` and sent to a meta object method as an argument. The meta object is one of the internal or external objects (meta object and its method are specified in the filter element that accepted the message). Within the meta object method, it is possible to use three specific statements: `continue`, `reply`, and `send`. When continue statement is used, the reified message is reactivated and continues with the next filter in the filter set. When the reply statement is used, the reified message is no longer considered and the sender receives the argument of the reply statement. With the send statement, the reified message is reactivated (like `continue` statement) until it reaches the return statement, so that the meta object method can have access to the *return value* of the message. The `send` statement is followed by `reply` or `continue` statement. No substitution is carried out. **Reject**: The message continues with the next filter in the filter set.
`Wait`	**Accept**: The message continues with the next filter in the filter set. No substitution or delegation is carried out. **Reject**: The rejected message is blocked until the condition corresponding to the matching part who matched the message becomes true. The message is then re-evaluated by the wait filter.
`Dispatch`	**Accept**: If a new target and/or a new selector are specified in `MSPList`, they are substituted in the accepted message, and then, the message is sent to the new target. The remaining filters in the filter set are no longer considered. **Reject**: The message continues with the next filter in the filter set.
`Realtime`	**Accept**: The timing attributes of the accepted message are changed, and then, the message continues with the next filter. **Reject**: The message continues with the next filter in the filter set.

Table 1. The filter handler actions.

`Sender`, `Recipient` and `DeliveryAgent` are threads that coordinate their behavior using a `bounded buffer` synchronization schema. A `Sender` thread creates an `Email` object, fills in the content, indicates the address and puts it in the `DeliveryAgent` buffer. The latter stamps the

Filter type	Syntax used to specify the filter element	C value	T.S matches MSPList	Effect when ⇒ is used		Effect when ~> is used	
				on FE	on filter	on FE	on filter
Error	C ⇒ { $T_1.S_1$, …, $T_i.S_i$}Or C ~> { $T_i.S_i$, …, $T_i.S_i$} MSPList is {$T_1.S_1$, …, $T_i.S_i$} No substitution or delegation carried out	False	False/True	Reject	Continue	Reject	Continue
		True	False	Reject	Continue	Accept	Continue
		True	True	Accept	Accept	Reject	Reject
Meta	C ⇒ { $T_1.S_1(MO_1.MS_1)$, …, $T_i.S_i(MO_i.M_{Si})$ } MOi is the meta object and MSi one of its methods	False	False/True	Reject	Continue	~> is not used with meta filter	
		True	False	Reject	Continue		
		True	True	Accept	Accept		
Wait	C ⇒ { T1.S1, …, Ti.Si} Or C ~> { T1.S1, …, Ti.Si} No substitution or delegation carried out	False	False	Reject	Continue	Reject	Reject
		False	True	Reject	Reject	Reject	Continue
		True	False	Reject	Continue	Accept	Accept
		True	True	Accept	Accept	Reject	Continue
Dispatch	C ⇒{T1.S1(NT1.NS1),…,Ti.Si(NTi,NSi),}Or C ~> { T1.S1, …, Ti.Si,} NTi.NSi are the new target and the new selector	False	False/True	Reject	Continue	Reject	Continue
		True	False	Reject	Continue	Accept	Accept
		True	True	Accept	Accept	Reject	Reject
Realtime	C ⇒ { T1.S1(TC1), …, Ti.Si(TCi)} TCi is the timing constraint No substitution or delegation carried out	False	False/True	Reject	Continue	~> is not used with real time filter	
		True	False	Reject	Continue		
		True	True	Accept	Accept		

Table 2. Filter elements acceptance or rejection and their effect on the containing filter.

Email object, then delivers it to the Recipient buffer.

Threads that hold a reference on an Email object can invoke all its methods. Thus, the delivery agent can read the email content; the recipient can stamp the email itself etc. These manipulations do not correspond to a correct use of an email in the real world. An Email object should provide multiple views depending on the client type, i. e. the object issuing the call. For example, readContent() should not be available to the delivery agent, stamp(...) should not be available to recipient and sender, etc.

```
public class Email {
    public void modifyContent(String text){…}
// Modifies the content of the Email with a field of type String
    public String readContent(){…}
// Returns the Email content
    public Boolean deleteContent(){…}
// Deletes the Email content and returns a boolean as an indicator
    public void setRecipientAddress(Address C){…}
// Address in C becomes the recipient's address
    public Address readRecipientAddress(){…}
// Returns the recipient's address
    public void setSenderAddress(Address C){…}
// Address in C becomes the sender's address
    public Address readSenderAddress(){…}
// Returns the sender's address
    public Boolean modifyAttributes(Attributes A){...}
// A contains new attributes (urgency mark, number of tries, …)
    public void stamp(Date d, Time t){…} }
// Adds the date and time when t he Email was first processed
```

Listing 1. The Email class.

Multiple views problem has a suitable answer in CF by adding to class Email a CF interface which contains a set of filters enforcing the views.

Threads may invoke any Email method, but input filters reject or accept the message according to the sender type. Details of the CF interface are given in listing 2.

```
class Email interface {
internals   // No internals or externals needed in this case
externals
conditions
  private Boolean deliveryAgentView()  {…}
// Returns true if the client object is of type DeliveryAgent
  private recipientView()  {…}
// Returns true if  the client object is of type Recipient
  private Boolean senderView ()  {…}
// Returns true if the client object is of type Sender
inputfilters
  err : Error = {
      deliveryAgentView() ⇒ {readRecipientAddress,
                                  readSenderAddress, stamp},
      recipientView() ⇒ { readContent, deleteContent,
                  readSenderAddress, readRecipientAddress },
      senderView() ~> { stamp }  }
  delegate : Dispatch = {true => *(inner.*)}    /* All messages
that pass err are delegated to the kernel which is called inner */
}
```

Listing 2. A CF interface for class Email.

In this CF interface, filter err specifies that if the client is of type DeliveryAgent, only methods readRecipientAddress, readSenderAddress and stamp are allowed to pass to the next filter and then be dispatched to the kernel, otherwise, an exception is raised. The MSPList

of the filter `err` consists of only a list of selectors. If the client is of type `Sender`, all methods are accepted except `stamp`. Notice that this FE can be expressed using the inclusion operator and when ignoring inherited methods of class `Email`, as follows:

`senderView()` $\Rightarrow$ {`modifyContent, readContent, deleteContent, setSenderAddress, setRecipientAddress, readRecipientAddress, readSenderAddress, modifyAttributes`}

The `delegate` filter specifies that all messages, i. e. those having passed filter `err`, go to the kernel without changing their selectors.

3. The AspectJ Metamodel

3.1. Principle and Goals

Authors of this approach consider that the code of a given software system contains a base code part (i. e. the business logic) and a concerns part. The former consists of objects that achieve functionalities of the system, whereas concerns are made of code that crosscut functionalities, like transaction management and synchronization. In the object-oriented paradigm, concerns are scattered throughout the source code, consequently a tangling arises [12].

The AOP approach of ASPECTJ [13] proposes:

- To remove this tangling by separating the two parts of a software system: base objects and concerns which are called `aspects`.
- To weave the two parts by a tool called Weaver to get the executable code. The weaving is done using principled points of the code execution.

Hence, the vision motivating AOP is that one could provide independent specifications for each concern and functionality, and then weave them together to build the resulting system. The AOP approach is appropriate for a wide range of applications. It presents undeniable advantages in all applications where security, synchronization, transactions management, distribution, logging, etc, are considered as concerns and separated from the system base code. ASPECTJ aims at better modeling real world applications: code is smaller, less tangled and closer to our perception of the real world, and therefore enhances reuse, evolution and maintenance.

3.2. Basic Concepts

ASPECTJ is a general-purpose AOP extension to JAVA. It uses four new concepts: `Aspect`, `Pointcut`, `Advice` and `static crosscutting`. An aspect is an entity that looks like a class, but models a concern that crosscuts several object classes. To understand ASPECTJ approach, let us consider a system during its execution. At each moment, the system evolves toward new states while generating new facts and events like an access to, or change of a variable value, a method call, execution or return, etc. All these observable points called `Join points` represent a base code breaking points where aspects can get involved. Everything happens as if the execution of the base code was interrupted in the join points, giving a way to the aspect code to execute and enforce the concern purposes. Among all join points, only a subset may interest a given aspect. The aspect source code specifies its meaningful join points using `Pointcuts`. Pointcuts are particular forms of predicates that use boolean operators and specific primitives to capture join points and dynamic contextual information.

Several aspects can exist in the same software system and be concerned by the same join points and the same pointcuts. In this case, aspect must be composed using some precedence rules.

The aspect code is divided into blocks called `advices`. They are method-like mechanisms used to declare that a certain code should execute when join points in some pointcut are captured. Three possible relationships, that determine the advice type, bind advices to pointcuts: `before`, `after` and `around`. Thus, if a pointcut is a call to a method `m()`, then concerned aspects must specify a pointcut that captures the call join point and an advice that executes before or after the call of `m()` and even around it. The around advice can replace the call of `m()`, as it can specify instructions that execute before and others that execute after the call. This last possibility is specified by using a special statement called `proceed()` available only within the around advice body. Figure 3 outlines how advices preempt the normal execution flow according to the advice type. Figure 3a shows an

example of two objects where the method `m1()` of object `o1` calls the method `m2()` of object `o2`.

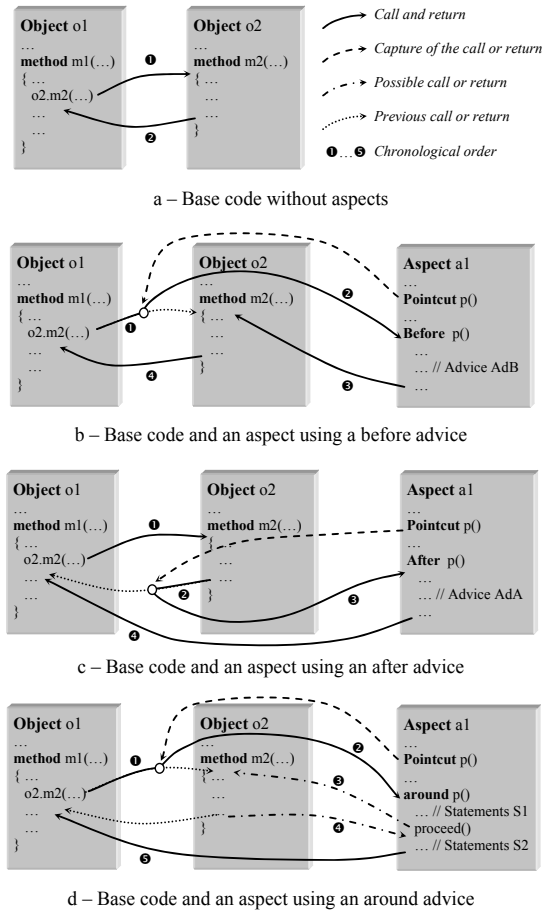


Fig. 3. Alteration of the control flow by an advice.

Figure 3b introduces an aspect `a1` that intercepts the call to `m2()` and executes some advice `AdB` and then resumes the call. Figure 3c shows an aspect that intercepts the control flow after the call to `m2()` is achieved and executes some advice `AdA`. In figure 3d, the aspect intercepts the call to `m2()`, executes some statements `S1`, resumes the call to `m2()` (using `proceed()`) and before returning to `m1()`, it executes some other statements `S2`.

ASPECTJ provides a rich set of primitive pointcuts to specify join points within an aspect. Table 3 summarizes the predicate meaning of those primitives and their composition.

One of the strengths of advices is that they can access the dynamic contextual information exposed by join points (this is called `context exposure`). For example, pointcuts can capture the parameters and the target object of a call. For this purpose, we can specify along with advices and pointcuts, parameters that will be bound to values of the current join points. Moreover, pointcuts can have an important descriptive power by using the wildcard "*" instead of a type, a method, a parameter or as a part of a name. In the same way, ".." is used to specify one or more parameters and "+" to include subtypes of a class within a specification.

Program	Comments

public class ComputeFactorial {
public static long factorial(int n) {
if (n==0) **return** 1;
else return n*factorial(n-1);} }
	Class `ComputeFactorial` uses the `factorial()`method to compute recursively the factorial of an integer n.

public aspect Optimizer { **pointcut** factOp(int n) : **call**(* *.factorial(int)) **&& args**(n);	The first pointcut of this aspect captures calls of `factorial()` along with its integer argument.
pointcut firstCall(int n) : factOp (n) **&& !cflowbelow**(factOp (int));	Captures only non-recursive calls of `factorial()` among those of pointcut `factOp()`. Ignoring recursive calls using `!cflowbelow()` allows us to cache only the final result.
private Cache fCache = **new** Cache(); /* `fCache` is private, only this aspect have access to it */	`fCache` is an object of class `Cache` that consists of a list of pairs `(int,long)`. The method `get(int)` returns a cached value and `put(int,long)` puts a value in the cache at a position corresponding to the first argument. To save space, this class is not shown.
before(int n) : firstCall (n) { **if** (n >50) bigValue();}	This before advice checks if the argument of the call is bigger than 50 then calls `bigValue()` which raises an exception. For short, this method is not shown.

long **around**(int n) : factOp (n) {
long val = fCache.get(n);
if (val != null) **return** val;
return proceed(n); }
	This around advice captures the call then checks the cache for the result value and return it if found. When not found, `proceed(n)` resumes the normal computation flow. Notice that when `proceed()` is not executed, the following advice (i.e. `after returning`) is ignored.

after(int n) **returning**(long result) : firstCall (n) { fCache.put(n, result); }}	This after returning advice captures the value returned at the end of the execution of `factorial()`. It specifies an argument to collect the return value in the `returning()` part. Then, it uses the context collected from the join point to update the cache.

Listing 3. Optimizing a computation by an aspect.

The last concept of ASPECTJ is the `static crosscutting` which modifies a model at compile time. For example, ASPECTJ allows specifying new members that are associated with other classes, as well as specifying what a class extends or implements. The former is called `introduction` or `inter-type member declaration`. Introductions allow an aspect to specify the concerned classes by using a prefix in the member declaration statement. When present, the prefix indicates one or more classes; otherwise the declared member is local to the aspect.

3.3. Example

Listing 3 shows an example of an aspect, adapted from [15], that optimizes the factorial computation. First, we have a simple class with a method called `factorial()` which computes recursively the factorial of an integer passed as an argument. Second, the aspect captures calls to `factorial()` and caches the computed values for latter use. Now, each time we want to compute a factorial by calling `factorial()`, the aspect captures the call and checks the cache for the result value and returns it if found. When the result is not in the cache, the normal computation flow is resumed. Notice that we can do the same thing by adding optimization code in the class `ComputeFactorial`, but such an implementation will tangle the optimization logic with factorial computation logic. By using an aspect, the optimization code is completely separated from the factorial computation code. This allows us to change the caching strategy without affecting the `ComputeFactorial` class.

4. Motivations

Aspect orientation and MDA can be related in various ways: such as considering aspects and business logic as two kinds of models or considering that a model is made up of aspects and some business logic, and so on [21]. The intent of this article is not to specify the aspect orientation/MDA relationship, but rather to focus on the transformation of two AOP approaches within the MDA context.

Pointcut	**Corresponding predicate value** (cjp stands for one of the current join points)	**Comments**
call(S)	True if cjp corresponds to a call to `S`	`S ::=ResultType ClassName.MethodName (Parameters)` to specify a method or `ClassName(Parameters)` to specify a constructor
execution(S)	True if cjp corresponds to an execution of `S`	
get(S)	True if cjp corresponds to an access to `S`	`S ::= Type ClassName.FieldName`
set(S)	True if cjp corresponds to an assignment to `S`	
initialization(S)	True if cjp corresponds to the execution of the initialization of an object in `S`	`S ::= ClassName(Parameters)` Parameters are those of the first constructor. Initialization includes the super constructor call
preinitialization(S)	True if cjp corresponds to the execution of the pre-initialization of an object in `S`	
staticinitialization(S)	True if cjp corresponds to the execution of the initialization of class `S`	`S ::= ClassName`
handler(TP)	True if cjp corresponds to the handling of the `TP` exception inside a catch bloc	`TP` specifies the exception type
within (TP)	True if cjp corresponds to the execution of a code belonging to `TP`	`TP` is a class name
withincode(S)	True if cjp corresponds to the execution of a code defined in a method or constructor specified by `S`	`S` is a method or constructor signature
cflow(P)	True if cjp is in the control flow of the join point defined by `P` (including `P` itself)	`P` is a pointcut
cflowbelow(P)	True if cjp is in the control flow below the join point in `P` (excluding `P` itself)	
adviceexecution()	True if the executing code belongs to an advice	
this(TP or Id)	True if cjp corresponds to the execution of a code belonging to the object defined by `TP` or `Id` (the object being the current object referenced by `this` in JAVA)	`TP` is a class name and `Id` an identifier. '..' replaces any number of parameters
target(TP or Id)	True if the target of cjp is an object specified by `TP` or `Id`	
args(TP or Id or '..')	True if the arguments of cjp are instances whose type is specified by `TP` or `Id`	
if(BoolExp)	True if `BoolExp` is true	`BoolExp` is a boolean expression
! P	True if `P` is not satisfied	`P`, `P1`, and `P2` are pointcuts
P1 && P2	True if both `P1` and `P2` are satisfied	
P1 || P2	True if `P1` or `P2` or both are satisfied	
(P)	True if pointcut between the brackets is satisfied	

Table 3. Predicate meaning of the primitive pointcuts and their composition.

Although CF and ASPECTJ are grouped within a same category (i. e. AOP approaches) they are different and both have strengths and weaknesses. We summarize some of their features in the following.

Description style. The CF allows expression of filters using a declarative style that is easy to understand and work with. In contrast, ASPECTJ has a procedural style, like JAVA, that is based on several new constructs to express concerns. The CF style is useful when dealing with delegation of messages and substitution. However, when more complex computations are needed, the ASPECTJ advices are better than the CF meta filter since they have a direct access to the dynamic contextual information.

Separation among concerns. In CF each filter in the filter set describes one concern. However, in some cases, two or more filters are combined to describe one concern. Consequently, a CF interface tends to describe concerns from the class point of view, i. e. the filter set gives all the concerns where the class is involved. In contrast, an aspect in ASPECTJ may describe a concern that involves several classes (i. e. a concern point of view). When modeling a system, we need the two points of view since sometimes we may focus on an isolated concern and sometimes we may focus on how concerns compose for a given class.

The join point model. ASPECTJ, is provided with a wealthy join point model that allows aspects to capture calls, access or modification of fields, exception handling and to specify concerns in various ways. On the opposite, CF intercepts only the incoming and outgoing calls.

The synchronization. While ASPECTJ adopts the JAVA synchronization mechanism, the CF offers a specific filter that allows the description of various synchronization concerns without dealing explicitly with the wait and the notify of threads.

According to the previous, concerns that are better expressed in CF are security, synchronization and timing constraints, while concerns that are better expressed in ASPECTJ are optimization and general control of computations, and control of access operations to object fields. Broadly speaking, CF is more appropriate for real time and concurrent applications since it deals well with timing constraints and concurrency using wait and real time filters [2]. In contrast, ASPECTJ is better when dealing with applications that need optimizing some computations, handling exceptions and field access control which are concerns we usually find in financial transaction systems, distributed systems, etc [9, 15]. However, in many cases applications have several kinds of aspects which justify the use of multiple AOP approaches.

Since CF and ASPECTJ are different and since metamodels influence the modeling task and the perception we have of the real world [8], it becomes advantageous to use them both to capture the real world subtle situations. This conclusion is the starting point of an ongoing work that aims to construct an MDA environment where software developers can use multiple AOP approaches at the same time and freely switch from one to another using automated tools that preserve the concerns' traceability.

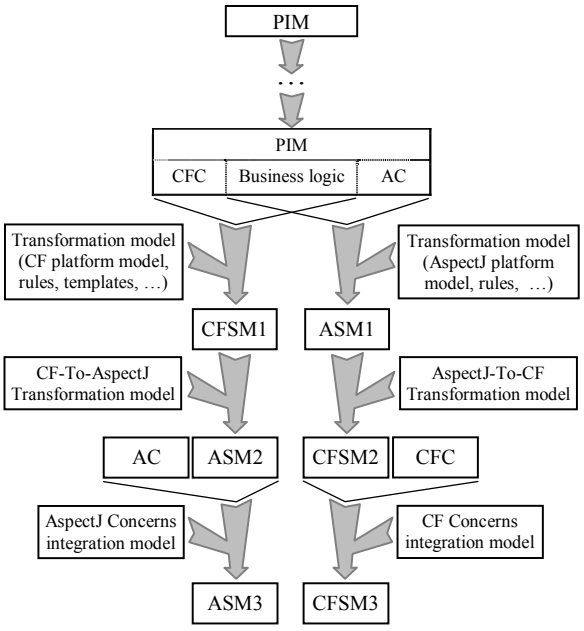


Fig. 4. Software development scenario.

Within this MDA environment, the software development scenario is depicted in Figure 4 as a cascade of PIM to PIM transformations, PIM to PSM transformations and PSM to PSM transformations. At the top, we find a use case PIM that highlights aspects of the system corresponding to the user view. Then, the use case PIM is

transformed (likely through a cascade of transformations) to extract objects, classes and concerns and get a PIM that expresses some business logic along with concerns that are platform independent. These concerns can be divided into two sets: concerns that are better expressed in CF (CFC) and concerns that are better expressed in ASPECTJ (AC).

In the next step, we get a CF specific model (CFSM1) or an ASPECTJ specific model (ASM1) by transforming the PIM using CF (resp. ASPECTJ) platform model, a set of rules, templates, etc. When the first CFSM (resp. ASM) is obtained, one can transform it to get ASM2 (resp. CFSM2) using the CF-To-ASPECTJ transformation (resp. ASPECTJ-To-CF transformation) and then add concerns that are better expressed in ASPECTJ (resp. in CF) to get the final PSM ASM3 (resp. CFSM3).

For example, consider an Email server system consisting of several services; one thread per user and a database where each user emails are stored. In this type of application we can describe the synchronization concern between threads using CF and the exception handling concern (transmission errors, timeouts, etc) using ASPECTJ.

Obviously, using the most suitable metamodel each time means that developing a whole system will involve several metamodels and transformations to get the final PSM. This drawback can be reduced by using automated tools and by preserving the traceability. In our context, preserving traceability is a challenging task since it means preserving a one to one correspondence between the CF concerns and ASPECTJ concerns. To achieve this goal, we need to convert concerns in CF models using as much as possible only concepts dedicated to express concerns in ASPECTJ and vice versa.

Allowing the use of CF and ASPECTJ metamodels during the development of a system is the first and the main motivation of this work. As a first step towards this ideal, this article is limited to only one way transformation of the CF models into ASPECTJ.

The second motivation arises from the fact that a transformation between two PSMs of the same abstraction level is useful in some cases. To understand this, let us consider the scenario of Figure 5.

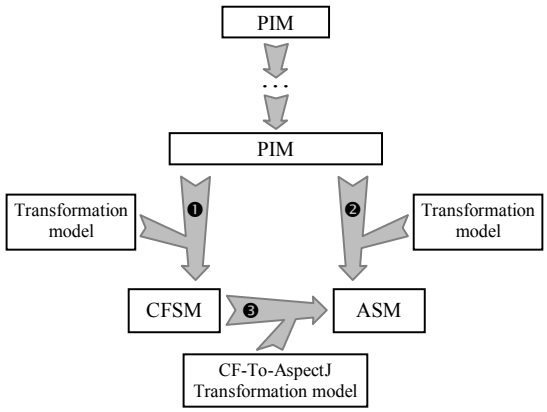


Fig. 5. Example of development scenario.

According to Figure 5, one can get the ASM using the transformation labeled ❷ or through transformation ❶ followed by transformation ❸ . This last, which is our concern, is useful in the following cases:

- We have the CFSM and want to get the ASM, unfortunately there is no PIM from which it can be generated. This situation is still frequent today. Indeed, the MDA well-defined style is not always followed and software developers still take a PSM-only approach where platform specificities are considered from the early development phases, and separately defined models are not used (i. e. PIM and PSM) [8]. Notice, in this case, that concerns are directly described in the PSM (CFSM or ASM).
- We would like to change the AOP platform (e. g. ASPECTJ instead of CF), but at the same time we do not want to spend an important effort in developing the transformation from the PIM to the new PSM. Since the two PSMs are relatively close, the PIM becomes useless and the ASM is advantageously generated from the CFSM.

The third motivation of this work lies in our confidence that through such PSM to PSM transformations and by preserving traceability, one can deduce common features of AOP approaches. These features are potential candidates for an AOP metamodel to build AOP PIMs. To the best of our knowledge, an AOP metamodel to express AOP PIMs does not exist yet and we are currently investigating this subject in another research.

5. CF to AspectJ Transformation

5.1. Overview

In the MDA context, the transformations can be from PIM to PIM, PIM to PSM, PSM to PSM or PSM to PIM, and several approaches can be used for transforming models such as the `marking`, the `model transformation`, or the `metamodel transformation`, etc (see [14, 18] for more details).

In this work, we propose a PSM to PSM metamodel transformation which can be illustrated using the MDA transformation representation of Figure 6. The transformation model consists of a set of normalizing rules and transformation templates (see Figure 7).

Since CF and ASPECTJ are both extensions of JAVA, the transformation can be done by converting the CF concepts using the JAVA part of ASPECTJ. But, when doing so, we achieve a simple weaving that does not preserve the traceability of concerns. In order to preserve the traceability, we need to use, as much as possible, only concepts dedicated to express concerns in ASPECTJ. For this sake, we keep the base code classes of the CF model unchanged and transform the CF interfaces using aspects, advices, pointcuts and static crosscutting.

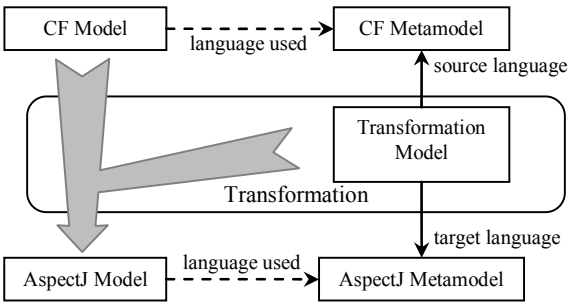


Fig. 6. PSM to PSM metamodel transformation.

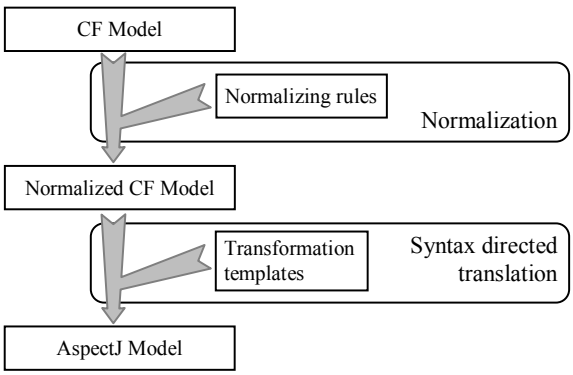


Fig. 7. Overview of the transformation.

The transformation we propose consists of two processes (Figure 7):

- A normalization that aims to put the filters in a canonical form which facilitates the transformation.
- A translation using a syntax-directed approach and transformation templates. It consists of generating a set of aspects using a syntax-directed approach guided by a set of templates that preserve the CF semantics.

Notice that some concepts of CF like realtime filter are currently outside the proposed transformation. The reasons motivating this restriction are given in subsection 5.5.

In the remainder of this section, we will illustrate the transformation using the example of listing 4, which includes four filter types and almost all different forms of filter elements. Some comments are given in the same listing.

5.2. Normalization

The goal of this process is to determine all the accessible methods of a CF class and to put its filter elements in a canonical form that facilitates the transformation. A CF interface adds the public methods of internal/external objects to the set of kernel class methods. In case where two or more methods have the same name, the one belonging to the kernel class (may be inherited) or the internal/external object declared first hides the others. Hence, the signature σC of an object `C` (or a class `C`) can be defined as the set of all methods accessible for incoming messages. Notice that incoming messages do not specify internal/external objects as targets. But their specifications in filters impose to the CF object to deduce which target will execute the message.

In the example of listing 4, σ`A=` {`inner.m1(), inner.m2(), int1.mb1(), int1.mb2(), int3.mc1(), int3.mc2(), int3.mc3(), ext1.md1()`}. Methods `int2.mb1()` and `int2.mb2()` are masked by `int1` since it is declared first. In the same way `ext1.m1()` and `ext1.m2()` are masked by the `inner`.

According to the previous, normalizing the methods' part in the example consists of replacing the list of declared methods with its complement to σ_A (i. e. σ_A-{`inner.m1()`, `inner.m2()`}, see listing 5).

To put filter elements in a canonical form we apply a set of normalizing rules so that:

- The dispatch or meta filter elements have the form:
  `C ⇒ selector(NewTarget.NewSelector)`
- The wait filter elements have the form:
  `C ⇒ Selector` where `C` is a condition or a conjunction of conditions
- The error filter elements have the form: `C ⇒ { Selector1, Selector2,..., Selectori}` or `C ⇒ Selector`

Filters' normalizing rules can be grouped in seven categories:

– Eliminating * within a `MSPList`:
  `T.*(NT.NS)` is equivalent to
  `T.S1(NT.NS), ..., T.Si(NT.NS)` **if** σ_T=`{S1, ..., Si}`
  `T.S(*.NS)` is equivalent to `T.S(inner.NS)`
  `T.*(NT.*)` is equivalent to `T.S1(NT.S1),..., T.Si(NT.Si)`
  if σ_T=`{S1, ..., Si}` and $\sigma_T \subset \sigma_{NT}$
  `T.*` without a substitution part, is equivalent to
  `T.S1, ..., T.Si` **if** σ_T=`{S1, ..., Si}`

– Eliminating exclusion operator:
  `C~>{T1.S1, ..., Ti.Si}` is equivalent to
  `C⇒` σ_{inner} `- {T1.S1, ..., Ti.Si}`

– Adding inclusion operator and condition:
  `{T1.S1, ..., Ti.Si}` is equivalent to `⇒{T1.S1, ..., Ti.Si}`
  `⇒{ T1.S1, ..., Ti.Si}` is equivalent to
  `True⇒{T1.S1, ..., Ti.Si}`

– Adding substitution part in dispatch filter elements:
  `S` is equivalent to `S(NT.S)` **if** `NT.S` ∈ σ_{inner}
  `S(NS)` is equivalent to `S(NT.NS)` **if** `NT.NS` ∈ σ_{inner}

– Decomposition (does not apply to error filters)
  `C⇒{T1.S1(NT1.NS1), ..., Ti.Si(NTi,NSi)}` is equivalent to
  `C⇒T1.S1(NT1.NS1), ..., C⇒Ti.Si(NTi,NSi)`
  `C⇒{T1.S1, ..., Ti.Si}` is equivalent to
  `C⇒T1.S1, ..., C⇒Ti.Si`

– Ignoring target in the matching part. Also applies when `T=inner`
  `T.S(NT.NS)` is equivalent to `S.(NT.NS)` **if** `T.S` ∈ σinner
  `T.S` is equivalent to `S` **if** `T.S` ∈ σinner

– Grouping wait filter elements having the same matching part
  `C1 ⇒S, C2 ⇒S, ..., Ci ⇒S` is equivalent to
  `{C1,C2,...,Ci}⇒S`

```
public class A {
    public void m1();
    public void m2();
    ... }
public class B {
    public void mb1();
    public void mb2();
    ... }
public class BB {
    public void mb1();
    public void mb2();
    ... }
public class C {
    public void mc1();
    public void mc2();
    public void mc3();
    ... }
public class D {
    public void m1();
    public void m2();
    public void md1();
    ... }
class A interface {
        //A is the kernel
internals
        public B int1;
        public BB int2;
        public C int3;
externals
        public D ext1;
conditions
private boolean cd1(){...}
private boolean cd2(){...}
private boolean cd3(){...}
private boolean cd4(){...}
methods
    void m1(); void m2();
```

```
Inputfilters
  err:    error = { cd1() ⇒ {int1.*,int3.*},          // err1
                    cd2() ~> {m2},                    // err2
                    {int3.mc1} }                      // err3
  act:    meta  = { cd3() ⇒ {m1(ext1.md1),int1.*(ext1.md1)} }   // act1
  sync:   wait  = { cd4() ⇒ {m2, int3.*},             // sync1
                    cd3()⇒ m2 }
  deleg:  dispatch={ cd1()⇒{m1(m2), int1.mb1(int3.mc2)},   // deleg1
                     cd2()~>{ext1.*},                 // deleg2
                     cd3()⇒{int1.*(int2.*)},          // deleg3
                     cd4()⇒{inner.*(ext1.*)} }        // deleg4
}

Comments
err1: When cd1 is true, calls to the methods of int1 and int3 are accepted and pass to filter act.
err2: When cd2 is true, all messages are accepted except m2 of the kernel
err3: Calls of mc1 in int3 are unconditionally accepted
act1: When cd3 is true, the calls of m1 in the kernel are reified and passed as argument to md1 in ext1, and when cd3 is true, calls to methods of int1 are reified and passed as argument to md1.
sync1: Causes the calls to m2 in the kernel to be blocked until cd4 and cd3 become true. The calls of all methods in int3 are also blocked until cd4 becomes true.
deleg1: When cd1 is true, calls of m1 in the kernel are dispatched to m2 and calls of mb1 in int1 are substituted by mc2 and delegated to object int3.
deleg2: When cd2 is true, all calls except those where target is ext1 are dispatched without substitution. This means that calls to md1 are not dispatched. The other methods, m1 and m2 in ext1, are not in the scope of incoming messages, since they are hidden by m1 and m2 of the kernel class A.
deleg3: When cd3 is true, the calls to methods of int1 are delegated to int2 without any substitution.
deleg4: When cd4 is true, all the calls to the kernel methods (m1 and m2) are dispatched to the external object ext1 without substitution.

Remarks: For brevity, methods do not return values and only the public methods are shown
The wild card '*' always matches the target or selector in the matching part and does not modify the target or the selector when specified in the substitution part.
When the target is omitted, the kernel object is considered instead.
When a condition is missing, true is assumed instead.
```

Listing 4. Example of a CF model.

When applying the normalizing rules to the example in listing 4, we get the results shown in listing 5.

```
methods     void mb1(); void mb2(); void mc1();
            void mc2(); void mc3(); void md1();
Inputfilters
err : error = {
    cd1()⇒ {mb1, mb2, mc1, mc2, mc3},          // err_1
    cd2()⇒ {m1, mb1, mb2, mc1, mc2, mc3, md1}, // err_2
    true⇒ mc1 }                                // err_3
act : meta = {
    cd3()⇒ m1(ext1.md1),                       // act_1
    cd3()⇒ mb1(ext1.md1),                      // act_2
    cd3()⇒ mb2(ext1.md1) }                     // act_3
sync : wait = {
    {cd4(), cd3()} ⇒ m2,                       // sync_1
    cd4()⇒ mc1,                                // sync_2
    cd4()⇒ mc2,                                // sync_3
    cd4()⇒ mc3 }                               // sync_4
deleg : dispatch ={
    cd1() ⇒ m1(inner.m2),                      // deleg_1
    cd1() ⇒ mb1(int3.mc2),                     // deleg_2
    cd2()⇒ m1(inner.m1),                       // deleg_3
    cd2()⇒ m2(inner.m2),                       // deleg_4
    cd2()⇒ mb1(int1.mb),                       // deleg_5
    cd2()⇒ mb2(int1.mb2),                      // deleg_6
    cd2()⇒ mc1(int3.mc1),                      // deleg_7
    cd2()⇒ mc2(int3.mc2),                      // deleg_8
    cd2()⇒ mc3(int3.mc3),                      // deleg_9
    cd3()⇒ mb1(int2.mb1),                      // deleg_10
    cd3()⇒ mb2(int2.mb2),                      // deleg_11
    cd4()⇒ m1(ext1.m1),                        // deleg_12
    cd4()⇒ m2(ext1.m2) }                       // deleg_13
```

Listing 5. Normalized methods and input filters parts.

5.3. Transformation Templates

Valid transformation changes the structure of the original model, but preserves its behavior (i. e. its semantics). Recall that a CF model is a set of ordinary JAVA interfaces and JAVA classes where some are provided with CF interfaces. Therefore, the overall structure transformation of the CF model is to keep unchanged ordinary JAVA classes and JAVA interfaces while translating CF interfaces into aspects. Consequently, the main transformation task is to find what corresponds to each CF interface.

Since the CF interface represents several concerns, it translates to several aspects. We propose the structure transformation schema of Figure 8.

According to Figure 8, the JAVA interfaces and classes are kept unchanged in the ASPECTJ model while each CF interface has a counterpart consisting of:

- One aspect for each filter in the input filter set.
- One aspect composed of inter-type members' declaration introducing internal/external instance variables and public methods into the kernel class. These public methods have an empty implementation body since the corresponding calls will be captured by aspects and delegated to internal/external objects.
- One aspect called `kernel_Final` to capture calls that are not dispatched and raise a corresponding exception.

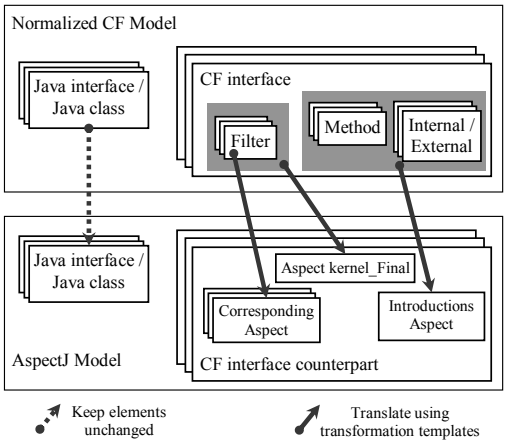


Fig. 8. The structure transformation schema.

The content of these aspects is determined using the transformation templates given in the rest of this subsection. Listing 6 shows the resulting model when applying the structure transformation schema to the example of listing 5.

Aspects that correspond to filters are composed of advices. For example, a normalized FE `C⇒S(NT.NS)` in a filter maps to an advice in the corresponding aspect. This advice is of type `around` and has three anonymous pointcuts (i. e. unnamed pointcuts directly declared in the advice). The first pointcut is `call(* * inner.S())` which captures calls to method S in the inner class (the first wildcard means any access modifier and the second any return value type). The second and third pointcuts are `target(obj)` and `args(parameterList)`. They make available the target object and arguments of the call within the advice body which uses them to perform its computation.

```
public class A {      public void m1();
                      public void m2(); … }
public class B {      public void mb1();
                      public void mb2(); … }
public class BB {     public void mb1();
                      public void mb2(); … }
public class C {      public void mc1();
                      public void mc2();
                      public void mc3(); … }
public class D {      public void m1();
                      public void m2();
                      public void md1(); … }
privileged aspect A_Introd {
      declare precedence A_err, A_act, A_sync, A_deleg, A_final;
      public B A.int1;
      public BB A.int2;
      public C A.int3;
      public D A.ext1;
      public boolean A.cd1(){…};
      public boolean A.cd2(){…};
      public boolean A.cd3(){…};
      public boolean A.cd4(){…};
      public void A.mb1(){};
      public void A.mb2(){};
      public void A.mc1(){};
      public void A.mc2(){};
      public void A.mc3(){};
      public void A.md1(){};
}
privileged aspect A_err { …}        // See listing 7
privileged aspect A_act { …}        // See listing 8
privileged aspect A_sync { …}       // See listing 9
privileged aspect A_deleg { …}      // See listing 10
aspect A_final { …}                 // See listing 11
```

Listing 6. Transforming the structure of the CF model.

Preserving the CF semantics is a challenging task. If we consider a filter set `FS` where each filter `F` is composed of filter elements `FElem`, then preserving the CF semantics means: implementing the accept and reject effects of each `FElem` on messages and on the filter `F`, and implementing the accept and reject effects of each filter `F` on messages and filter set `FS`.

Particularly, the following constraints must be satisfied:

① A message passes once through each filter in the filters set and each FE in an order that corresponds to their declaration. Up-down for filter set and from left to right for filter elements.

② According to the filter type, the filter element handler action may affect the message flow. After a dispatch, a final reject or reification, the remaining FE and filters are no longer considered. In the case of a wait filter, each time the message is blocked by a filter element, it passes the filter again.

③ A message must be dispatched by the filter set.

These constraints can be expressed in terms of filter element accept or reject effects on the control flow of a message according to the filter type, the position of the filter element within the filter and the position of the filter within the filter set (see Table 4).

Filter Type	Current FE handler Action	Is current FE the last FE in the filter?	Is current filter the last filter in the filter set?	Effect on the message flow within a normalized input filter set
Error	Accept	No⁄Yes	No	Pass to next filter
			Yes	Raise an exception
	Reject	No	No⁄Yes	Pass to next filter element
		Yes	No⁄Yes	Raise an exception
Meta	Accept	No⁄Yes	No	If `m()` executes *Continue* or *send*, then pass to next filter `m()` is the method to which the reified message is sent
			Yes	If `m()` executes *Continue* or *send*, then raise an exception
	Reject	No	No⁄Yes	Pass to next filter element
		Yes	No	Pass to next filter
			Yes	Raise an exception
Wait	Accept	No	No⁄Yes	Pass to next filter element
		Yes	No	Pass to next filter
			Yes	Raise an exception
	Reject	No⁄Yes	No	Block the message then pass the current filter again
			Yes	Raise an exception
Dispatch	Accept	No⁄Yes	No⁄Yes	Dispatch the message. Ignore remaining filter elements and filters
	Reject	No	No⁄Yes	Pass to next filter element
		Yes	No	Pass to next filter
			Yes	Raise an exception

Table 4. Effect of filter element handler actions on the message flow.

To enforce constraint ① in ASPECTJ, we declare precedence between aspects. The aspect corresponding to the first filter will dominate the aspect corresponding to the second that will dominate the aspect corresponding to the third and so on. The FEs' order is enforced by declaring the corresponding advices in the same order of the FEs in the filter.

To enforce the second constraint, we remark that since to each FE corresponds an advice, ignoring the remaining FEs and filters in case of a dispatch or reject comes to inhibiting their corresponding advices. 'Inhibiting' means attaching a condition to an advice, to preclude its execution. For this sake, we have added a class, called `List`, whose objects are inhibition lists. Objects of class `List` contain inhibition

elements which consist of a join point identifier, a target object identifier and a string that specifies if the inhibition concerns the whole input filter set or only a given filter.

When processing a filter element, `CFE`, which causes the filter `F` to accept a message and skip the remaining FEs in `F`, the advice corresponding to `CFE` inserts an element (`joinpoint_id, target_id, ''filterName''`) in the inhibition list. In the case of a dispatch or reification with `reply`, the inserted element is (`joinpoint_id, target_id,''filterSet''`) which inhibits all the remaining advices and aspects corresponding to the filter set. All advices must check for the absence of inhibiting elements before proceeding.

Advice handles the inhibition list by three methods, `inhibit()`, `removeInhibition()` and `inhibited()`, which respectively inserts, removes elements, and tests for the existence of elements in the list. The reflexive features of JAVA and ASPECTJ allow us to know, at any time, the current join point and current object unique identifiers by using `thisJoinPoint`.

Notice that a form of inhibition already exists in ASPECTJ for advices of type around. Indeed, when an around advice executes, it precludes the execution of all other advices of lower

Aspect corresponding to Error filter err	**Comments**
privileged aspect A_err { **object around** (): … { …} // corresponds to err_1	The aspect name is composed of the kernel class name and the filter name.

object around (A obj): // corresponds to err_2
(**call**(* * A.m1(..)) || **call**(* * A.mb1(..)) ||
call(* * A.mb2(..)) || **call**(* * A.mc1(..)) ||**call**(* * A.mc2(..)) ||
call(* * A.mc3(..)) ||**call**(* * A.md1(..))) &&**target**(obj) {
if ((! list.inhibited(tjp, obj, "filterset")&&
(! list.inhibited(tjp, obj, "A_err")) &&
cd2()){ list.inhibit(tjp, obj, "A_err");
}
return proceed();
}
	Anonymous point cuts corresponding to the matching part.
	This around advice tests if all the aspects corresponding to the filter set have not been inhibited by a previous aspect, that advices of `A_err` have not been inhibited by a previous advice and that `cd2()` is true. If so, the call is accepted and the remaining advices in `A_err` are inhibited. The `proceed()` allows the call to pass to other advices. `tjp` stands for `thisJoinPoint`.

object around (): … { …} // corresponds to err_3	
object around (A obj): **call**(* * A.*(..)) && **target**(obj) { **if** (!list.inhibited(tjp, obj, "A_err")) raiseException(); **else** { list.removeInhibition(tjp, obj, "A_err"); **return** proceed();} } }	At the end of the aspect, an advice is added to raise an exception if the call has not been accepted by the previous advices in `A_err`. If the call has been accepted, the inhibition element of this aspect is removed from the list. The `proceed()` allows the call to pass to advices of next aspect.

Listing 7. Error filter transformation template.

Aspect corresponding to Meta filter act	**Comments**

privileged aspect A_act {
object around(): … { …} // corresponds to act_1
	The second advice tests if `A_act` and aspects corresponding to the filter set have not been inhibited and if `cd3()` is true. If so, aspects corresponding to this filter and filter set are inhibited.
	During the transformation, statements of the meta object method `md1()` are changed and inlined within the advice body. Recall that there are three special statements `send`, `reply` and `continue` that can be applied to a reified message `mes` within meta object method. Occurrences of '`mes.continue()`' statement are replaced with :
	`List.removeInhibition(tjp, obj,"filterset");`
	`return proceed(paramList);`
	Occurrences of '`mes.reply(expression)`' are replaced with:
	`return expression`
	Occurences of '`variable = mes.send()`' are replaced with:
	`List.removeInhibition(tjp, obj,"filterset");`
	`variable = proceed(paramList)`
	Notice that access to the arguments `paramList` can be done directly since they are not reified. Removing the filter set inhibition allows the call to pass through aspects corresponding to the remaining filters.
	Within the advice, any reference to a field or a method of the meta object is prefixed by '`this.ext1.`'

object around(A obj, paramDecl): // corresponds to act_2
(**call**(* * A.mb1(..)) && **target**(obj) &&
args(paramList) {
if ((! list.inhibited(tjp, obj, "filterset")&&
(! list.inhibited(tjp, obj, "A_act"))) && cd3(){
list.inhibit(tjp, obj, "filterset");
list.inhibit(tjp, obj, "A_act");
…
… // inline here instruction of md1()
} **else return** proceed();
}
}	

object around(): … { …} // corresponds to act_3
}
/* *paramDecl* stands for a parameter declaration list and *paramList* for a list of parameters */	

Listing 8. Meta filter transformation template.

Aspect corresponding to Wait filter sync	Comments

privileged aspect A_sync {
  private Lock A_ed4_ed3;
  private Lock A_ed4;
object around (A obj): // corresponds to sync_1
  call(* * A.m2(..))&& **target**(obj) {
  if ((! list.inhibited(tjp, obj, "filterset") {
  while (!(ed4() && ed3())) A_ed4_ed3.block();
  }
  return proceed();
}
object around (): … { …}// corresponds to sync_2
…
after (): **set**(* A.fie|d1) ‖ **set**(* A.fie|d2) …{
  A_ed4_ed3.wakeup();
}
after (): **set**(* A.fie|d2) ‖ **set**(* A.fie|d4) …{
  A_ed4.wakeup();
}
}
	The overall transformation consists of adding a new class called *Lock*, having two synchronized methods *block()* and *wakeup()* whose role is respectively to block threads using the JAVA *wait()* statement and resume them using the JAVA *notifyAll()* statement.
	We attach a lock to each condition or conjunction of conditions in a filter element. Each call to a message that matches a FE in MSPList causes the thread to test the condition and block repeatedly until the condition becomes true.
	At each iteration, the thread is resumed by an *after* advice that captures *set* join point on any of the variables composing the condition. Then the thread evaluates the condition again. When a thread leaves the while loop, it executes *proceed()* which allows the advices in the remaining aspects to alter the message flow.
	
	Notice that no filter inhibition is necessary since pointcuts in each advice concern different calls (see grouping wait filter elements normalization rule).
	In the same way, the filter set is not inhibited since wait filters do not alter the message flow but just delay it.
	field1, field2, etc are assumed to be the fields that appear in the method bodies of ed4() and ed3(). Notice that these fields can be determined by parsing ed4 and ed3 condition methods.

Listing 9. Wait filter transformation template.

Aspect corresponding to Dispatch filter deleg	Comments

privileged aspect A_deleg {
object around(): … { …} // corresponds to deleg_1
object around(A obj, paramDecl): // corresponds to deleg_2
  call(* * A.mb1(..)) && **target**(obj) && **args**(paramList){
  if ((! list.inhibited(tjp, obj, "filterset") && ed1()) {
  list.removeInhibition(tjp, obj, "filterset");
  return this.int3.me2(paramList);
  } **else return** proceed();
}
…
object around(A obj, paramDecl): // corresponds to deleg_4
  call(* * A.m2(..)) && **target**(obj) && **args**(paramList){
  if ((! list.inhibited(tjp, obj, "filterset") && ed2()) list.inhibit(tjp, obj, "filterset");
  return proceed();
}… }
	An around advice inhibits all the remaining around advices unless proceed is used. Thus two cases are possible. In the first, when the message is to be dispatched to the kernel object with the same selector, we use `proceed()` to let the message continue but, at the same time we need to inhibit all the remaining advices, that's why we use `list.inhibit(tjp, obj, "filterset")`.
	In the second case, when message is dispatched to internal/external objects or the kernel class with a new selector, we remove the inhibition element which is no longer needed and then delegate the message. The remaining advices are implicitly inhibited since we do not use `proceed()`.

Listing 10. Dispatch filter transformation template.

precedence concerned by the same join points, except when `proceed()` is used. In this case, the next advice with the highest precedence is executed. In our transformation, we rely on the ASPECTJ inhibition whenever possible and use `proceed()` whenever we want to avoid it. In listings 7 to 10, we take a closer look to the transformation templates of each filter type. For clarity sake, we use the normalized filter examples of listing 5 rather than a specific formal template language.

Finally, to enforce constraint ③, we add a special aspect called `kernel_Final` with only one advice that captures any call to the kernel class and raises an exception unless it is inhibited (see listing 11).

```
aspect A_Final {
  object around(A obj): call(* * A.*(..)) && target(obj) {
    if ((! list.inhibited(tjp, obj, "filterset") raiseException();
    else { list.removeInhibition(tjp, obj, "filterset");
     return proceed(); }
 }}  /* raiseException is a method that throws a
NotDispatched exception */
```

Listing 11. Contents of the kernel_final aspect.

5.4. The Syntax-directed Transformation

The second process in the transformation consists of translating the normalized CF models into ASPECTJ. For this sake, we use the syntax-directed translation approach described in [1]

and the transformation templates given in 5.3. Formally, a syntax-directed translation schema is a 5-tuple T = (N, Σ, Δ, R, S) where:

- N is a finite set of nonterminal symbols
- Σ is a finite input alphabet
- Δ is a finite output alphabet
- R is a finite set of pairs of rules having the form (A→ α, A≡ β) where the first is a derivation rule and the second is the corresponding translation element. A∈N, $\alpha \in (N \cup \Sigma)^*$ and $\beta \in (N \cup \Delta)^*$. α and β are strings composed of terminals and nonterminals so that the nonterminals in β are a permutation of the nonterminals in α. → is the derivation symbol and ≡ is the translation symbol.
- S is a distinguished nonterminal in N called start symbol

When transforming a model `M1` (having `MM1` as a metamodel) to another model `M2` (with `MM2` as a metamodel), the syntax-directed translation is used as a method of transforming derivation trees in the input metamodel `MM1` into derivation trees in the output metamodel `MM2`. Given an input sentence `x`, a translation for `x` is obtained by constructing a derivation tree for `x`, then transforming the derivation tree into a tree in `MM2`, and then taking the frontier of the output tree as a translation for `x`. The transformation of a normalized CF model can be characterized by the set of rules given in Table 6, which include the structure transformation schema and the templates. Sets N, Σ and Δ are defined as follows.

- N = {ENTITY, REMAINDER, CFINTERFACE, CD, VD, ...} : all words given in small capital letters
- Σ = {**class**, **interface**, **internals**, **conditions**, ∼>, ⇒, **(,),** ...} **:** all keywords and terminal symbols used in CF are given in bold typeface
- Δ = {**class**, **interface**, **aspect**, **privileged**, **pointcut**, **before, after, around, inhibited, obj, (,),** ...} : all keywords and terminal symbols used in ASPECTJ are given in bold typeface

In Table 6, strings in italic typeface are inserted, as they are, in the generated model and '_' indicates a concatenation. The notation A→ $\{\alpha \mid \beta\}$ and A≡ $\{\chi \mid \delta\}$ represents two rules (A→ α, A≡ χ) and (A→ β, A ≡ δ). Words beginning by $ are directives used to localize positions in the translated model where additional statements are inserted. The statements corresponding to each directive are given in Table 5.

Instead of	Insert
$Precedence	Precedence declaration statement
$LastAdvice	Advice of listing 11, with $Kernel instead of A
$EndFilterError	The last advice of listing 7 with $Kernel instead of A
$EndFilterWait	*After* advices of listing 9 with $Kernel instead of A and variables used in filter element conditions instead of field1, field2, …
$Kernel	The name of the kernel class
$ParamDecl	List of parameters of the concerned method along with their types
$ParamList	List of parameters of the concerned method
$MetaMethod	The body of the corresponding meta method after its transformation according to listing 8
$Locks	As many lock declarations as conditions in the considered wait filter
$LockName	The name of the lock corresponding to the condition
$Filter	The name of the corresponding filter

Table 5. Directives and their corresponding codes.

5.5. Scope of the Transformation

Concerning the transformation of the CF models to ASPECTJ, we have proposed an approach that covers all CF interface constructs including four filter types. Currently, the `Realtime` filter type is not supported since ASPECTJ does not provide specific concepts to deal with timing constraints. Building a real time executive subsystem in ASPECTJ is possible, but it will be mainly based on JAVA.

CF has rich syntactical forms to specify filters. Even if some forms have not been considered in the given examples, they can be normalized and the same transformation applied. Output filters are not covered by the transformation for the reasons given in 2.2, but we expect that most of the proposed transformation can be reused. Superimposition, which is a technique introduced in CF to deal with systemic crosscutting (see [6] for more details), is not covered. This is due to the lack of complete and well documented semantics and the lack of an implementation that covers CF features and superimposition in the same time. The last implementation of CF (i. e. extension of the Microsoft DotNet platform) does not support superimposition [6].

6. Related Work

Although AOP and MDA are powerful paradigms, they are still relatively new and little has been done in relating them. Most related work deal with model transformations or the role of AOP in MDA. However, to the best of our knowledge, there is no attempt in transforming one AOP approach using another within the MDA context and even outside it. In the following, we first present a work on the taxonomy of model transformation approaches, then we present works dealing with the transformation of AOP approaches and finally we present some works dealing with the role of AOP in MDA.

Production rule	Corresponding translation element
S → ENTITY REMAINDER	S ≡ ENTITY REMAINDER
REMAINDER → ENTITY REMAINDER	REMAINDER ≡ ENTITY REMAINDER
ENTITY → { JAVACLASS | JAVAINTERFACE |CFINTERFACE }	ENTITY ≡ { JAVACLASS | JAVAINTERFACE | CFINTERFACE }
CFINTERFACE → **class** IDENTIFIER **interface** { INTERNALSPART EXTERNALSPART CONDITIONSPART METHODSPARTS INPUTFILTERPART }	CFINTERFACE ≡ **aspect** IDENTIFIER_*Introd* { $Precedence INTERNALSPART EXTERNALSPART CONDITIONSPART METHODSPARTS } INPUTFILTERPART **aspect** IDENTIFIER_*Final* {$LastAdvice }
INTERNALSPART → **internals** VARIABLEDECL	INTERNALSPART ≡ VARIABLEDECL
VARIABLEDECL → { VD | VD**;** VARIABLEDECL }	VARIABLEDECL ≡ { VD | VD**;** VARIABLEDECL }
VD → **public** TYPEIDENTIFIER IDENTIFIER	VD ≡ **public** TYPEIDENTIFIER**.** IDENTIFIER
EXTERNALSPART → **externals** VARIABLEDECL	EXTERNALSPART ≡ VARIABLEDECL
CONDITIONSPART → **conditions** CONDITIONDECL	CONDITIONSPART ≡ CONDITIONDECL
CONDITIONDECL → { CD | CD**;** CONDITIONDECL }	CONDITIONDECL ≡ { CD | CD**;** CONDITIONDECL }
CD → **private** RETTYPE IDENTIFIER **( )** { METHODBODY }	CD ≡ **public** RETTYPE IDENTIFIER **( )** { METHODBODY }
METHODSPART → **methods** METHODDECL	METHODSPART ≡ METHODDECL
METHODDECL → { MD | MD**;** METHODDECL }	METHODDECL ≡ { MD | MD**;** METHODDECL }
MD → **public** RETTYPE IDENTIFIER **( )**	MD ≡ **public** RETTYPE IDENTIFIER **( ) { }**
INPUTFILTERSPART → *inputfilters* FILTERDECL	INPUTFILTERSPART ≡ FILTERDECL
FILTERDECL → { FD | FD FILTERDECL }	FILTERDECL ≡ { FD | FD FILTERDECL }
FD → IDENTIFIER **:** FILTERSPECIF	FD ≡ **privileged aspect** $Kernel_IDENTIFIER **{** FILTERSPECIF
FILTERSPECIF → { **error = {** ERRORFEDECL **} | meta = {** METAFEDECL **} | wait = {** WAITFEDECL **} | dispatch = {** DISPFEDECL **}** }	FILTERSPECIF ≡ { ERRORFEDECL $EndFilterError } | METAFEDECL } | $Locks WAITFEDECL $EndFilterWait } | DISPFEDECL } }
ERRORFEDECL → { ERRORFE| ERRORFE**,** ERRORFEDECL }	ERRORFEDECL ≡ { ERRORFE | ERRORFE ERRORFEDECL }
ERRORFE → CONDITION ⇒ LISTSELECTOR	ERRORFE ≡ **object around (**$Kernel *obj***): LISTSELECTOR &&target(***obj***) { if ((!** *list.inhibited***(thisJoinPoint,** *obj***, "***filterset"***)&& (!** *list.inhibited***(thisJoinPoint,** *obj***, "**$Kernel_**$**Filter"**)) &&** CONDITION**){** *list.inhibit***(thisJoinPoint,** *obj***, "**$Kernel_**$**Filter"**); } return proceed(); }**
LISTSELECTOR → { SELECT | **{LS}** }	LISTSELECTOR ≡ { SELECT | **(**LS**)** }
LS → { SELECT | SELECT**,** LS }	LS ≡ { SELECT | SELECT**,** LS }
SELECT → SELECTOR	SELECT ≡ **call(* *** $KerneL**.**SELECTOR **(..)**
METAFEDECL → { METAFE | METAFE**,** METAFEDECL }	METAFEDECL ≡ { METAFE | METAFE METAFEDECL }
METAFE → CONDITION ⇒ SELECTOR **(**TARGET**.**NEWSELECTOR**)**	METAFE ≡**object around($Kernel** *obj***,** $ParamDecl**): (call(* * $Kernel.**SELECTOR**(..)) && target(***obj***) && args(**$ParamList**) { if ((!** *list.inhibited***(thisJoinPoint,** *obj***, "***filterset"***)) && (!** *list.inhibited***(thisJoinPoint,** *obj***, "$Kernel_$Filter"))) &&** CONDITION **{** *list.inhibit***(thisJoinPoint,** *obj***, "***filterset"***);** *list.inhibit***(thisJoinPoint, obj, "**$Kernel_$Filter**");** $MetaMethod **} else return proceed(); } }**
WAITFEDECL → { WAITFE | WAITFE**,** WAITFEDECL }	WAITFEDECL ≡ { WAITFE | WAITFE WAITFEDECL }
WAITFE → COND ⇒ SELECTOR	WAITFE ≡ **object around (**$Kernel *obj***): call(* *** $Kernel.**SELECTOR(..)) && target(***obj***) { if ((!** *list.inhibited***(thisJoinPoint,** *obj***, "***filterset"***) { while (!** COND**)** $LockName.*block*();**} return proceed();}**
COND → { CONDITION | **{LC}** }	COND ≡ { CONDITION | **(**LC**)** }
LC → { CONDITION | CONDITION**,** LC }	LC ≡ { CONDITION | CONDITION **&&** LC }
DISPFEDECL → { DISPFE | DISPFE**,** DISPFEDECL }	DISPFEDECL ≡ { DISPFE | DISPFE DISPFEDECL }
DISPFE → { CONDITION ⇒ SELECTOR **(inner.**SELECTOR**)** | CONDITION ⇒ SELECTOR **(inner.**NEWSELECTOR**)** | CONDITION ⇒ SELECTOR **(**TARGET**.**NEWSELECTOR**)** }	DISPFE ≡ **object around(**$Kernel *obj***,** $ParamDecl**): call(* *** $KerneL**.**SELECTOR **(..)) && target(***obj***) && args(**$ParamList**) { if ((!** *list.inhibited***(thisJoinPoint***, obj, "filterset"***)** CONDITION**) { list.inhibit(***thisjoinpoint***,** *obj***, "***filterset"***); return proceed(); } | { list.removeInhibition(thisJoinPoint,** *obj***, "***filterset"***); return this.**NEWSELECTOR **(**$ParamList**);} else return proceed(); } \| { list.removeInhibition(thisJoinPoint,** *obj***, "***filterset"***); return this.**TARGET**.**NEWSELECTOR **(**$ParamList**);} else return proceed(); }**
CONDITION → { **true** | IDENTIFIER **()** }	CONDITION ≡ { **true** | IDENTIFIER **()** }
JAVACLASS, JAVAINTERFACE, METHODBODY have the same syntax as those of JAVA. IDENTIFIER, TYPEIDENTIFIER, RETTYPE, SELECTOR, NEWSELECTOR, TARGET are lexical variables	JAVACLASS, JAVAINTERFACE, METHODBODY are kept unchanged IDENTIFIER, TYPEIDENTIFIER, RETTYPE, SELECTOR, NEWSELECTOR, TARGET are lexical variables

Table 6. CF to ASPECTJ transformation rules.

Concerning model transformation approaches, a taxonomy is provided in [16]. The purpose of this taxonomy is to give an overview of the research field of model transformation and to present a framework for comparing and combining transformation tools, techniques and formalisms. The authors identify a set of criteria to characterize a transformation approach such as: number of source and target models, technical space of the transformation (e. g., MDA, XML), endogenous/exogenous (i. e. transformation within the same language or between languages), horizontal/vertical (i. e. transformation within the same level of abstraction or across levels of abstraction), syntactic/semantic (i. e. simple syntactical rewriting or complex transformation taking semantics into account), etc. According to some of these taxonomy criteria, our transformation approach can be characterized as a horizontal exogenous semantic transformation in the MDA and AOP technical space.

Concerning the transformation of AOP approaches, most works deal with the weaving of business logic and concerns to get a target program in a pure object-oriented language. This is especially the case of CF, ASPECTJ and hyperspace approach (HyperJ) for which the weaving produces a JAVA program (see [11] and the special issue of CACM [9]). In contrast, our work deals with the transformation of an AOP approach (CF) using another (ASPECTJ). Although this can be considered as a special weaving, it is, however, achieved under the strong constraint of preserving business logic classes unaltered (i. e. preserving the concerns traceability).

Concerning the role of AOP in MDA, the work described in [21] advocates the fact that AOP addresses a fundamental problem faced by the MDA: how to define separate models for concerns and business logic, combine those models, and finally generate applications from them? In the same way, we find in [10] a presentation showing a particular integration between Cosmic and C-SAW. The former is a tool suite, consisting in modeling languages and platform specific generative tools, dedicated to distributed real-time and embedded systems. The latter is an aspect model weaver which is applied to Cosmic models to transform them according to the characteristics of crosscutting modeling concerns. The authors highlight the capability of the resulting environment to support quick insert/remove of new properties and policies into a model without extensive manual adaptation. Compared to these attempts, our work, in its current state, does not address the problems that may arise when integrating AOP and MDA, however, we suppose that an AOP PSM contains business logic and concerns that have been derived from a set of PIMs without constraining their forms or contents.

7. Conclusion

Both aspect orientation and MDA are powerful paradigms whose combination is a promising issue for software development and maintenance. In this article we focused on the transformation of CF models into ASPECTJ models considered as two PSMs. The strength of our work lies in two points: the first is that we propose a syntax-directed transformation for which the automated tool, that constructs and traverses derivation trees, is easily implemented, second is the exclusive use of concepts that extend JAVA in ASPECTJ. This avoids our transformation to be a simple weaving that produces a JAVA model. The focus on how aspect-oriented concepts are related enhances the abstraction level of the transformation and preserves traceability of concerns between the two models.

In our future work, we will consider the inverse transformation (i. e. ASPECTJ to CF), along with transformations between other SOC approaches. Ultimately, we will exploit these transformations to deduce high level aspect-oriented concepts that are platform independent.

8. Acknowledgment

We would like to thank the anonymous referees whose comments helped us to significantly improve the first version of this article.

References

[1] A. V. AHO ET AL, *Compilers. Principles, Techniques and Tools*, Addison Wesley, 1986.

[2] M. AKSIT, B. TEKINERDOGAN, *Solving the Modeling Problems of Object-Oriented Languages by Composing Multiple Aspects Using Composition Filters,* AOP'98 workshop position paper, 1998. `http://trese.cs.utwente.nl/composition_filters/`

[3] M. AKSIT, B. TEKINERDOGAN, *Aspect-Oriented Programming Using Composition Filters*, ECOOP'98 Workshop Reader, Springer Verlag, July 1998. http://trese.cs.utwente.nl/composition_filters/

[4] L. BERGMANS, *Composing Concurrent Objects, Ph.D. thesis,* University of Twente, 1994. http://trese.cs.utwente.nl/composition_filters/

[5] BERGMANS L. , M. AKSIT, *Composing Crosscutting Concerns Using Composition Filters*, Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.

[6] L. BERGMANS, M. AKSIT, *Principles and Design Rationale of Composition Filters*, In Aspect-Oriented Software Development, Addison-Wesley, 2004. http://trese.cs.utwente.nl/publications/files/0169Chapter4CompFilts.pdf

[7] J. BÉZIVIN, *From Object composition to Model Transformation with the MDA*, in proceeding of TOOLS'USA, Vol. IEEE TOOLS-39, Santa Barbara, August 2001.

[8] A. BROWN, *An introduction to Model-Driven Architecture, Part I: MDA and today's systems,* http://www-06.ibm.com/developerworks/rational/library/3000.html

[9] T. ELRAD ET AL., *Aspect-Oriented Programming*, Special theme, Communications of the ACM, Vol. 44, No. 10, October 2001.

[10] J. GRAY, A. GOKHALE, *Concern Separation in Model-Integrated Computing*, OMG's First Annual Model-Integrated Computing Workshop, Arlington, VA USA, October 12–15, 2004.

[11] E. HILSDALE, J. HUGUNIN, *Advice Weaving in* ASPECTJ, 3rd International Conference on Aspect-Oriented Software Development, pp. 26–35, April 2004.

[12] G. KICZALES ET AL., *Aspect-Oriented Programming*, in Proc. of ECOOP'97, Lecture Notes in Computer Science Vol. 1241, pp. 220–242, 1997. http://eclipse.org/ASPECTJ

[13] G. KICZALES ET AL, *An Overview of* ASPECTJ, in Proc. of ECOOP, Springer-Verlag, 2001.

[14] A. KLEPPE ET AL, *MDA explained: The Model-Driven Architecture Practice and Promise*, Addison Wesley, 2003.

[15] R. LADDAD, ASPECTJ *in action: Practical Aspect-Oriented Programming*, Manning Publications Co., 2003.

[16] T. MENS, P. VAN GORP, *A Taxonomy of Model Transformation, International Workshop on Graph and Model Transformation, Tallinn, Estonia,* September 28, 2005. ftp://ftp.umh.ac.be/pub/ftp_infofs/2005/GraMOT-taxonomy.pdf

[17] J. MILLER, J. MUKERJI, *Model Driven Architecture (MDA), Document number ormsc/2001-07-01, Architecture Board ORMSC*, July 9, 2001. http://www.omg.org/mda/presentations.htm

[18] J. MILLER, J. MUKERJI, *MDA Guide version 1.0.1, OMG,* June 2003. http://www.omg.org/mda

[19] *Meta Object Facility Specification,* Version 1.4, Object Management Group, April 2002. http://www.omg.org/mda

[20] H. OSSHER, P. TARR, *Multi-Dimensional Separation of Concerns using Hyperspaces.* IBM Research Report 21452, April, 1999.

[21] D. WAMPLER, *The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture, Aspect Programming, Inc.,* http://www.aspectprogramming.com/papers, Last visited August 2005.

Received: January, 2005
Revised: August, 2005
Accepted: December, 2005

Contact address:

Djamel Meslati
Laboratoire de Recherche en Informatique
Université de Annaba
BP 12
23000, Annaba
Algérie
e-mail: meslati_djamel@yahoo.com

Mohamed T. Kimour
Laboratoire de Recherche en Informatique
Université de Annaba
BP 12
23000, Annaba
Algérie
e-mail: kimour@yahoo.com

Saïd Ghoul
Computer Science Department
Philadelphia University
Sweilah PoBox 1101
Amman
Jordan
e-mail: sghoul@philadelphia.edu.jo

DJAMEL MESLATI is the head of the research group on evolution and reuse of software systems. His research interests include software development and evolution methodologies and separation of concerns.

MOHAMED T. KIMOUR is an assistant professor at the department of computer science at the University of Annaba. His research interests include requirements engineering and model-based development of embedded real-time systems.

SAÏD GHOUL is a professor at the department of computer science at the Philadelphia University in Amman. His research interests include software process methodologies and programming languages.