INFORMATICA, 2009, Vol. 20, No. 4, 555–578

On ASPECTJ and Composition Filters: A Mapping of Concepts

Djamel MESLATI
LRI Laboratory, University of Annaba
BP 12, 23000 Annaba, Algeria
e-mail: meslati_djamel@yahoo.com

Received: 3 March 2006; accepted: 3 July 2007

Abstract. ASPECTJ and composition filters are well-known influential approaches among a wide range of aspect-oriented programming languages that have appeared in the last decade. Although the two approaches are relatively mature and many research works have been devoted to their enhancement and use in practical applications, so far, there has been no attempt that aims at comparing deeply the two approaches. This article is a step towards this comparison; it proposes a mapping between ASPECTJ and Composition filters that put to the test the two approaches by confronting and relating their concepts. Our work shows that the mapping is neither straightforward nor one-to-one despite the fact that the two approaches belong to the same category and provide extension of the same Java language.

Keywords: aspect-oriented programming, ASPECTJ, composition filters, mapping of concepts, separation of concerns, weaving.

1. Introduction

Both ASPECTJ (Kiczales *et al.*, 1997) and composition filters (CF) (Aksit and Teknerdogan, 1998a), are among the most well-known and mature aspect-oriented programming (AOP) approaches available today. They seek new modularizations of software systems by providing better concepts and mechanisms to appropriately separate concerns. Although a large amount of literature is devoted to these approaches in particular and separation of concerns (SOC) in general, so far, there has been no attempt that aims at deeply comparing the two approaches. We believe that a direct comparative study which relates the concepts of the two approaches will be helpful for their enhancement and emergence of a unified approach to separation of concerns. As a step towards this goal, we have achieved a mapping between ASPECTJ and CF which provide for each approach a particular translation that uses the concepts of the other. In a previous work we have focused on the mapping from CF to ASPECTJ in a model-driven architecture context (Meslati *et al.*, 2006). In this article, our focus will be on a reverse translation that maps the ASPECTJ concepts using the Composition Filters ones.

The motivation of the mapping between ASPECTJ and Composition Filters is twofold:

- The first is to directly relate concepts of the two approaches which can be used as a basis to achieve a comparative study. The comparative study, which is out of the scope of this article, is worthwhile and has clear motivations such as enhancing both CF and ASPECTJ, helping developers in choosing one approach or another, developing new hybrid approaches with a synergy of CF and ASPECTJ features, etc.

- The second is to allow the use of multiple SOC approaches in a model integrated computing (MIC) environment (Meslati *et al.*, 2006; Heidenreich *et al.*, 2009). The model integrated computing 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 (Gray and Gokhale, 2004). Broadly speaking, MIC is an approach where models are first class entities. A software system can be seen as a collection of models of various abstraction levels (requirement, design artefact or even a program which has the salient feature to be executable) where each describes the system from some viewpoint and, consequently, most engineering tasks can be considered as modelling and transforming models (Kleppe *et al.*, 2003). MIC and SOC can be related in various ways and their integration is a promising issue. Since each SOC approach has its own philosophy and concepts, an approach might be suitable from some viewpoint but inappropriate from another. Consequently, providing a MIC environment where multiple SOC approaches can be used simultaneously is a worthy goal. This means that a developer may build, for instance, a model containing all the concerns that are easily expressed in composition filters (e.g., synchronization), then transform this model to an ASPECTJ one and finally add concerns that are better expressed in ASPECTJ (e.g., exception handling). To achieve this goal we need first considering each SOC language as a metamodel, and then supply transformations (or mappings) between these metamodels. The use of multiple SOC approaches in a MIC environment is a motivation that gives to the mapping proposed in this article an intrinsic value.

The rest of this article is composed of five sections. Section 2 describes briefly the ASPECTJ approach. Section 3 presents the CF approach and describes its concepts in depth. Section 4 summarizes how CF concepts are mapped in ASPECTJ and then describes the mapping from ASPECTJ to CF. Section 5 discusses related work, and Section 6 is devoted to a conclusion and future work.

2. The AspectJ Approach

In many software applications, significant concerns are not easily expressed in a modular way. Examples of such concerns are synchronization, security and persistence. The code addressing these concerns is often scattered all over the application parts. ASPECTJ provides explicit language support for modularizing application concerns that crosscut the application base code. By separating the base code from crosscutting concerns (called aspects), the application source code becomes untangled and consequently becomes easy to understand, maintain and reuse. To obtain the executable code, a special tool called weaver is used to combine the application base code and its specific aspects (Kiczales *et al.*, 1997, 2001).

ASPECTJ is a general-purpose AOP extension to Java that introduces four concepts: `Aspect`, `Pointcut`, `Advice` and `static crosscutting`. An aspect is an entity that looks like a class but models a concern that crosscuts object classes. Pointcuts are declarations used in an aspect to identify principled points in the program execution and source code locations where it can be involved. Principled points such as an access or change of a field value, a method call or a method execution are called `Join points`. Pointcuts are particular forms of predicates that use boolean operators and specific primitives to capture join points and dynamic contextual information such as parameters of a call statement. ASPECTJ supports eleven different kinds of join points: method call, method execution, constructor call, constructor execution, field get, field set, preinitialization, initialization, static initialization, handler, and advice execution join points. There are also nine kinded pointcut designators that match join points according to their kind: `call, execution, get, set, preinitialization, initialization, staticinitialization, handler`, and `adviceexecution`.

The aspect code is divided into blocks called advices. They are method-like mechanisms used to declare that a certain code should execute before, after or around the code corresponding to the join points captured by some pointcuts. Therefore, there are three possible relationships that bind an advice to pointcuts: before, after and around. ASPECTJ provides a rich set of primitive pointcuts to specify join points within an aspect; see Meslati *et al.* (2006) or Laddad (2003) for more details.

The last concept of ASPECTJ is the `static crosscutting` which modifies a program at compile time by specifying new members of a class (called `introduction`) or specifying what a class extends or implements (called `inter-type member declaration`).

3. The Composition Filters Approach

CF uses the conventional object model and considers an object as an entity that performs some assigned functions (Aksit and Tekinerdogan, 1998a; Bergmans, 1994b). Within a system, entities interact with each other to achieve a common task. In the object model, most interactions are done by sending and receiving messages, that is where CF intervenes. It provides an object (which is then called Kernel) with an interface containing filters that intercept and manipulate messages in various forms modifying their scope and expected behaviours. The former consists of delegating messages to other objects (i.e., changing targets), whereas the latter consists of substituting the message selectors. By controlling messages (changing their targets and/or selectors) and through a well-constructed interface, CF provides suitable solutions to many problems (Aksit and Tekinerdogan, 1998b). One of the CF approach strengths is the use of a uniform filtration mechanism to resolve these problems. From this viewpoint, CF is easy to understand and work with as it only adds few concepts to the object model.

CF has been implemented as extension to several object-oriented languages and even as an extension to the .Net platform (Garcia, 2003). The CF syntax used in this article is that of ComposeJ which extends Java (Wichman, 1999). We justify this choice by the fact that when dealing with the same language extensions (i.e., Java), we focus more on the added concepts (i.e., those which deal with concerns) rather than the extended language specific features.

CF adds to an object a wrapping layer called interface that traps incoming and outgoing messages. Fig. 1 depicts the contents of an interface added to a kernel object.

An interface is composed of the following parts:

- **Internal objects** are combined with the kernel object to compose the state of the CF object. A message 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:** 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 (Bergmans, 1994b).
- **Input filters:** A set of declarative specifications that handle the incoming messages.

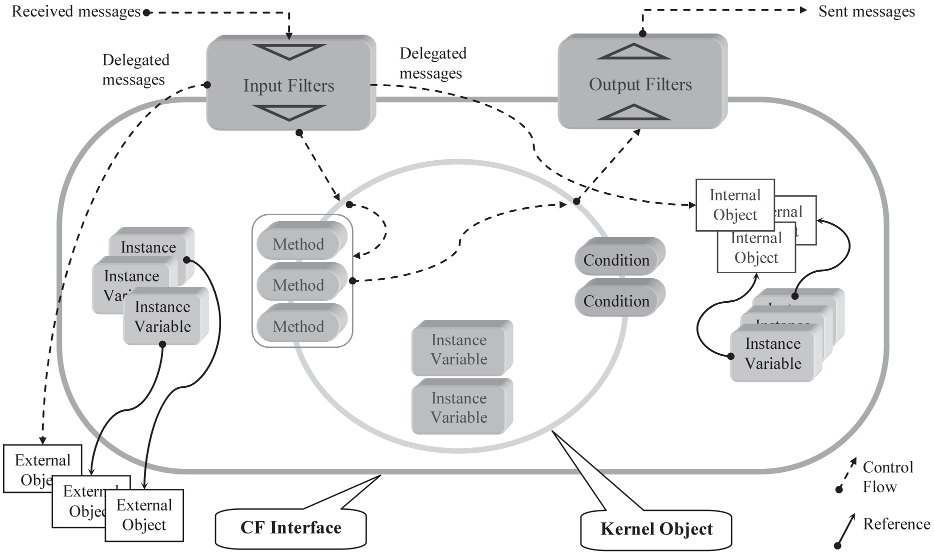

Fig. 1. Adding a CF interface to an object.

- **Output filters:** A set of declarative specifications that handle the outgoing messages.

The signature of a CF-object is the set of message selectors that the object would accept if all the conditions in the input filters would be true (Koopmans, 1995; see the description of filter elements in the next page). This includes public methods of the kernel class and public methods of the internal/external objects. 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 item 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. A call is also discarded when it passes the last filter in the filter set without being 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 by a call to another method in the kernel or internal/external objects and activating it.

Before discarding or dispatching a call, each filter through which the call passes can accept or reject it. Depending on the semantics of the filter type, "accept" may consist in dispatching or simply ignoring the message which then, passes to the next filter and "reject" may consist in discarding a message, queuing it as long as the filter expression results in a rejection or merely ignoring it (i.e., the message continues with the next filter, see Fig. 2). There are five commonly used filter types: `Error, Dispatch, Wait, Meta`, and `Realtime` (Bergmans and Aksit, 2001). 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 the other filter types to allow modelling 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, object oriented

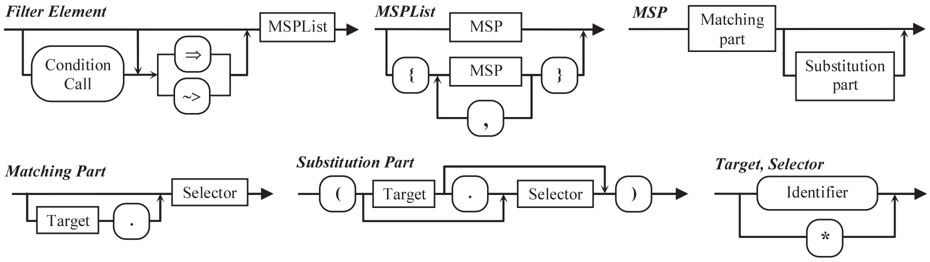

Fig. 2. Syntactical diagrams of filter elements (Meslati *et al.*, 2006).

paradigms tend 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., modelled in an input filter set rather than several output ones; Bergmans, 1994b). 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) and 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 Fig. 3). 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`. Fig. 2 depicts the syntactical diagrams corresponding to different forms of filter elements.

To simplify the filter set specification, CF proposes two operators ⇒ and ∼> called respectively inclusion and exclusion operators. `C` ⇒ `MSPList` means that when condition `C` is true, all messages that match `MSPList` will be accepted. `C` ∼> `MSPList` means that when the condition `C` is true, all messages, but those in `MSPList`, will be accepted. Notice

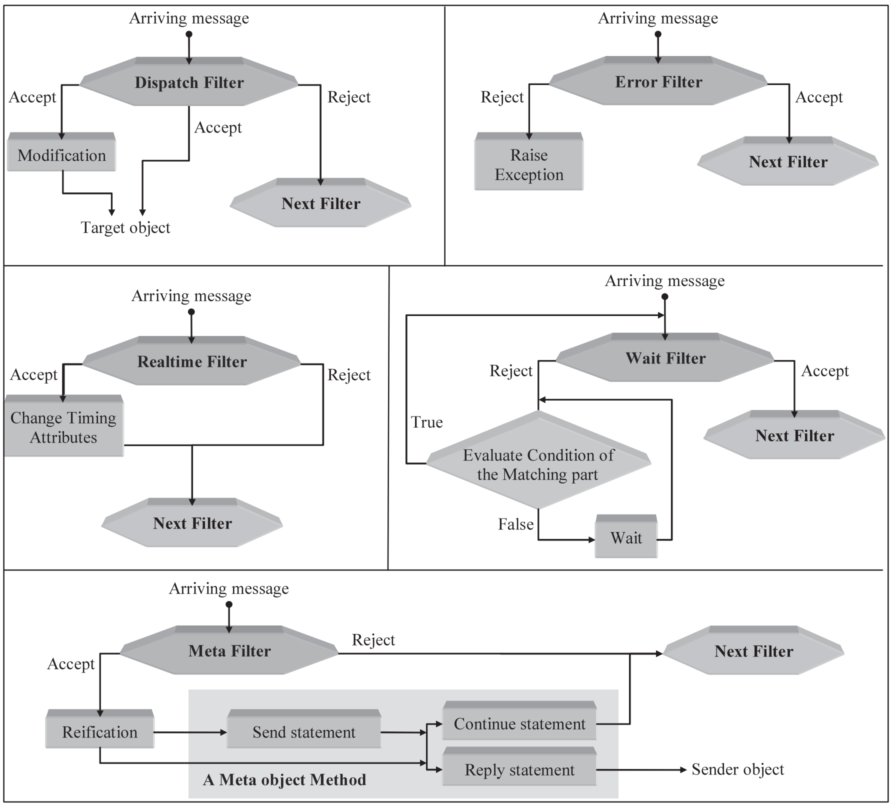

Fig. 3. The filter handler actions.

that the exclusion operator is not used for meta and realtime filters. In the same way, no substitution is carried out when the exclusion operator is used.

Fig. 3 summarizes the acceptance and rejection meaning for a filter according to its type. When a Dispatch filter accepts a message and 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. If a Dispatch filter action result in a rejection, the message continues with the next filter in the filter set.

With an `Error` filter, the accepted message continues with the next filter in the filter set and in case of a rejection an exception is raised.

The accept action of the `Wait` filter allows a message to pass to next filter in the filter set without achieving any substitution or delegation. When a message is rejected, it 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.

If a `Realtime` filter accepts a message it changes the attributes of the accepted message, which then continues with the next filter. In case of a rejection, the message continues with the next filter in the filter set.

In a `Meta` filter, the accepted message is reified as an object of special class called 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` (Koopmans, 1995). 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. Substitutions are not carried out with `meta` filters.

Table 1 shows when a filter element accepts or rejects a message and the effect on the filter level according to the condition value and the matching result; see Koopmans (1995) and Bergmans (1994a) for more details. We consider that an incoming message has `T` as target object and `S` as selector.

According to the value of the condition `C` of the FE, 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.

As an example, suppose that we have the following Dispatch FE `C` ⇒ {`T1.S1 (NT1.NS1)`, …, `Ti.Si(NTi,NSi)`, …}. If the condition `C` is false, then the call `T.S` is rejected by the FE and will continue with the next FE in the current filter. At the opposite, if the condition is true, then the handler actions depend on the matching process. If there is a matching of `T.S` with one term in the `MSPList {T1.S1(NT1.NS1)`, …, `Ti.Si(NTi,NSi)`, …}, then the FE accepts the call and, in turn, the current filter

Table 1

Filter elements acceptance or rejection and their effect on the containing filter (Meslati *et al.*, 2006)

Filter type	**Syntax used to specify the filter element** (T and S stand for the target of an incoming message and its selector respectively. C is the condition of the filter element)	**C value**	**T.S matches MSPList**	**Effect when $\Rightarrow$ is used**		**Effect when $\sim>$ is used**	
				On FE	**On filter**	**On FE**	**On filter**
Dispatch	$C \Rightarrow \{T_1.S_1(NT_1.NS_1), \ldots,$ $T_i.S_i(NT_i, NS_i), \ldots\}$ or $C \sim >\{T_1.S_1, \ldots, T_i.S_i, \ldots\}$	False	False/ true	Reject	Continue	Reject	Continue
	MSPList is $\{T_1.S_1(NT_1.NS_1), \ldots,$ $T_i.S_i(NT_i, NS_i), \ldots\}$ or $\{T_1.S_1, \ldots, T_i.S_i, \ldots\}$	True	False	Reject	Continue	Accept	Accept
	$NT_i.NS_i$ stands for the new target and the new selector	True	True	Accept	Accept	Reject	Reject
Error	$C \Rightarrow \{T_1.S_1, \ldots, T_i.S_i, \ldots\}$ or $C \sim> \{T_1.S_1, \ldots, T_i.S_i, \ldots\}$	False	False/ true	Reject	Continue	Reject	Continue
	No substitution or delegation	True	False	Reject	Continue	Accept	Continue
	are carried out	True	True	Accept	Accept	Reject	Reject
Meta	$C \Rightarrow \{T_1.S_1(MO_1.MS_1), \ldots,$	False	False	Reject	Continue	$\sim>$ is not used	
	$T_i.S_i(MO_i.MS_i), \ldots\}$	True	True	Reject	Continue	with meta filter	
	MO_i is the meta object and	True	False	Accept	Accept		
	MS_i one of its methods		True				
Wait	$C \Rightarrow \{T_1.S_1, \ldots, T_i.S_i, \ldots\}$ or	False	False	Reject	Continue	Reject	Reject
	$C \sim> \{T_1.S_1, \ldots, T_i.S_i, \ldots\}$	False	True	Reject	Reject	Reject	Continue
	No substitution or delegation	True	False	Reject	Continue	Accept	Accept
	are carried out	True	True	Accept	Accept	Reject	Continue
Realtime	$C \Rightarrow \{T_1.S_1(TC_1), \ldots,$ $T_i.S_i(TC_i), \ldots\}$	False	False/ true	Reject	Continue	$\sim>$ is not used with real time	
	TC_i is the timing constraint.	True	False	Reject	Continue	filter	
	No substitution or delegation are carried out	True	True	Accept	Accept		

accepts and dispatches it to the new target given in the term which matches T.S. If no term in MSPList matches T.S then the FE rejects the call and the process continues with the next FE in the current filter.

Now, suppose that we have the Dispatch FE C $\sim>$ { T1.S1, …, Ti.Si, …} and C is true. If one term of MSPList { T1.S1, …, Ti.Si, …} matches the call T.S then the FE rejects the call and, in turn, the current filter rejects it, which means that the message continues with the next filter in the filter set. If there is no match, then the FE accepts the call and the current filter accepts and dispatches it to the target object T with the same selector S.

4. The Mapping of Concepts

The purpose of the mapping is to answer the question: Given an ASPECTJ (or CF) program, what is the corresponding program in CF (respectively ASPECTJ)? Since the two languages are extensions of Java, and hence are Turing machine equivalent programming languages, we don't expect them to fail to express each other.

The challenge is rather to avoid that the translation be an ordinary weaving which produces a Java program. Indeed, by doing so, the resulting system is a tangled code that precludes any comparison. What we want is to use as much as possible only concepts dedicated to express concerns in each language. For instance, what filters correspond to an aspect? What pointcuts correspond to a condition in a FE? Since our focus is on how concerns are mapped, we constrained ourselves to preserve the base code unchanged during the mapping.

Due to the above constraint, some concepts of CF and ASPECTJ like realtime filter, get(), set(), are left unmapped. In the following, we describe shortly the CF to ASPECTJ mapping; please see Meslati *et al.* (2006) for more details, then we give a complete description of the inverse mapping.

4.1. *From CF to* ASPECTJ

The mapping from CF to ASPECTJ consists of two processes, normalization and translation using a syntax-directed approach (Fig. 4). The goal of the normalization process is to determine all the accessible methods of a CF class and put its filter elements in a canonical form to facilitate the subsequent translation.

The normalization process translates filters to a canonical form so that:

- dispatch or meta filter elements have the form:
  `C ⇒ selector(NewTarget.NewSelector)`

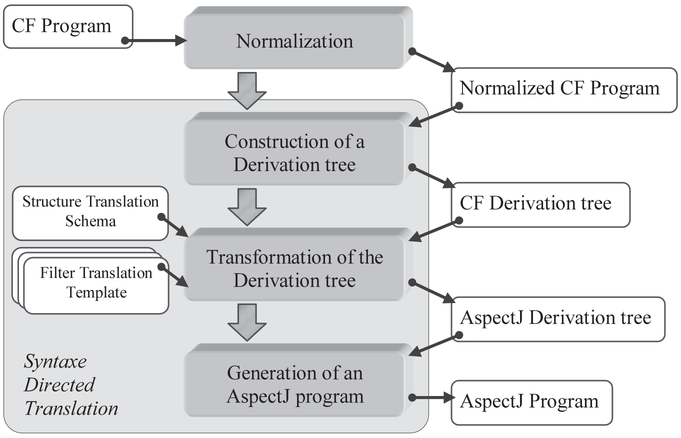
CF Program
Normalization
Normalized CF Program
Construction of a Derivation tree
CF Derivation tree
Structure Translation Schema
Filter Translation Template
Transformation of the Derivation tree
AspectJ Derivation tree
Syntaxe Directed Translation
Generation of an AspectJ program
AspectJ Program

Fig. 4. CF to ASPECTJ translation steps.

- wait filter elements have the form: `C ⇒ Selector` where `C` is a condition or a conjunction of conditions
- error filter elements have the form:
  `C ⇒ { Selector1, Selector2, ..., Selectori}`
  `or C ⇒ Selector`

There are seven normalizing rules to deal with various filter forms:

1) eliminating '*' within a MSPList;
2) eliminating exclusion operator;
3) adding inclusion operator and condition;
4) adding substitution part in dispatch filter elements;
5) decomposition (doesn't apply to error filters);
6) ignoring target in the matching part;
7) grouping wait filter elements having the same matching part.

The second translation process consists of translating the normalized CF programs into ASPECTJ. For this sake, we use the syntax-directed translation approach described in Aho *et al.* (1986) along with four filter `translation templates` and a `structure translation schema`.

The syntax-directed translation is used as a method of transforming derivation trees in the input grammar `G1` into derivation trees in the output grammar `G2`. 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 `G2`, and then taking the frontier of the output tree as a translation for `x` (see Fig. 5). The translation of a normalized CF program is characterized by a set of rules providing for each production rule a corresponding translation element.

The translation process is complex and the syntax directed approach alone is not suffi-

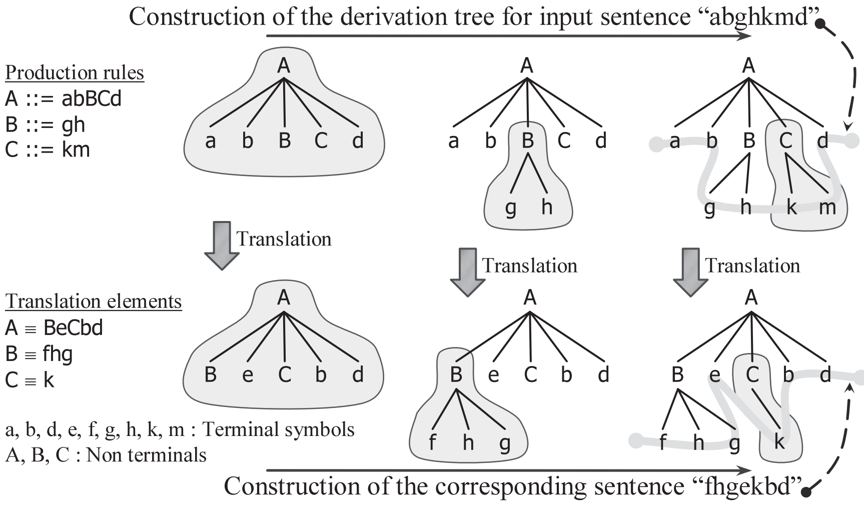

Fig. 5. Example of a syntax directed translation.

cient to generate an output ASPECTJ program that preserves the input CF program semantics. For this reason, the translation process uses two artefacts: filter translation templates and the structure translation schema.

In Meslati *et al.* (2006), we have proposed four filter translation templates that correspond to Error, Meta, Wait and Dispatch filters. Each template gives exactly what corresponds to a given filter in a form of translation elements and directives. Hence, the syntax directed translation gives as an output artefact an ASPECTJ program which contains terminal symbols and directives, then, the directives are converted according to the templates in a second phase.

In addition, the overall structure of a CF program is translated using a Structure translation schema. In this schema, Java interfaces and classes are kept unchanged in the final ASPECTJ program while for each CF interface there is: one aspect for each filter in the input filter set, one aspect called `kernel_final` (which captures non-dispatched calls and raises an exception) and one aspect composed of inter-type members' declaration introducing internal/external instance variables and public methods into the class that corresponds to 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.

From the behaviour point of view, 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., declared in the advice without names). The first pointcut is `call`(* * `inner.S`()) which captures calls to `S` method in the inner class (the first wildcard means any access modifier and the second, any return value type). The second and third pointcut are `target(obj)` and `args(parameterList)` that make available the target object and arguments of the call, which the advice body can use to perform its computation.

The translation of the CF programs to ASPECTJ covers all CF interface constructs including four filter types. `Realtime` filter type is not supported by the mapping since ASPECTJ doesn't provide specific concepts to deal with timing constraints. Concerning the normalization process, it can be extended merely by adding new rules, to deal with new filter forms.

Superimposition, which is a technique introduced in CF to deal with systemic crosscutting (see Bergmans and Aksit (2004) for more details), has not been covered in this work. This is due to the fact that the superimposition introduces an important complexity in the mapping as well as the lack of an implementation for Java that covers all the CF concepts and the superimposition at the same time.

4.2. *From* ASPECTJ *to Composition Filters*

The mapping of ASPECTJ programs into CF programs must deal with various concepts and three main features:

① Static crosscutting: It mainly consists of specifying members that are added to classes as well as specifying what a class extends or implements. These specifications change the base code classes at compile time.

② Dynamic crosscutting: It aims at changing the program behaviour at runtime by using the join point model, advices, context exposure and precedence between advices.

③ Aspect instantiation: Instance of an aspect can be attached to the whole program, to only one object or to the control flow of some pointcut.

In the following, we first present an overview of the mapping and give the algorithm that implements it, assuming that there is only one instance of each aspect and that each pointcut is anonymous and includes either a `Call()` or `Execution()` primitive pointcut. Second, we give an example of translation, and third, we generalize the translation approach through discussions on how particular concepts are translated or justify why they are not supported.

4.2.1. *Overview*

The overall mapping consists of transforming aspects, by projecting them on each base code class, in order to find CF interfaces in terms of internal/external objects, and input filters. The main idea is to get what is done by aspect advices done by external object methods and to translate what an aspect introduces, in a class `C`, into an internal class whose objects are associated with those of `C`. Fig. 6 shows an overview of the mapping of an ASPECTJ program into a CF program when considering one class and one related aspect.

To understand this mapping, consider an aspect `As` composed of `<IntertypeDecl, LocalDecl, Adv>`, where `IntertypeDecl` are introductions of members, `LocalDecl` are the aspect local members, and `Adv` are advices. The counterpart of `As` in CF, considering the base code class `C`, is a CF interface and two classes called `C_As` and `As`.

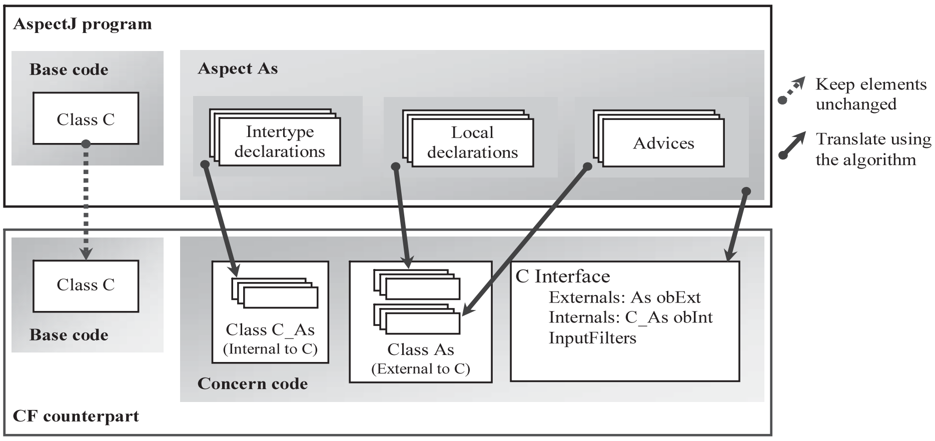

Fig. 6. Overview of the mapping.

- Class `C_As` contains `IntertypeDecl` members that are introduced in `C`. Since these members are specific to each object of `C`, we associate to each object of `C` an object of `C_As`. Therefore, we declare within the internals part of the CF interface of `C` one internal object of type `C_As`.
- Class `As` contains all `LocalDecl` members and for each advice in `Adv`, it contains one method whose name is composed from the advice type name and a discriminating number. Since we have only one instance of aspect `As`, we have also only one instance of class `As`, which is shared by all objects concerned by aspect `As`. Therefore, we declare within the externals part of the CF interface of `C` one external object of type `As`.

Each pointcut `Pc` associated with an advice can be decomposed into two parts: static and dynamic. Static part can be evaluated at compile time whereas the dynamic part is evaluated at runtime. For example, in the pointcut `(call(* *.factorial(int)) && !cflowbelow(call(* *.factorial(int))))`, the static part is `call(* *.factorial(int))` and the dynamic part is `!cflowbelow(...)`. Depending on the user specification, a pointcut can be only static or only dynamic.

In our mapping, the static part of an advice pointcut is used to determine which class and which method is concerned by the advice, and is no longer needed after that. The dynamic part becomes a condition that is evaluated by the method corresponding to the advice before executing.

The CF interface of class `C` contains, for each pair `(m, Ad)`, where `m` is a method in the base code class `C` and `Ad` an advice whose pointcut contains `call(m())` or `execution(m())`, a meta filter that reifies the call to m and sends it to a method `mAd`, in class `As`, corresponding to `Ad`. The second filter in the input filter set is the dispatch filter `true` ⇒ `*(inner.*)` which accept all incoming messages (i.e., accepts any call to methods of `C`). Notice that in some cases the pointcuts `call()` and `execution()` may not contain a class name, but just a method signature. In such cases, we need to parse the whole base code classes to find those having the methods that match the method signature in the pointcut.

4.2.2. *The Translation Algorithm*

Fig. 7 shows an algorithm implementing our mapping. The left part deals with aspect declarations that may contain introductions while the right part deals with advices and pointcuts.

To preserve ASPECTJ semantics, the program generated by the algorithm of Fig. 7 must deal with two specific features: advice precedence and access to members. Procedures that are concerned by precedence are: `Get next aspect`, `Get next advice`, `Add treatment corresponding to an advice` and `Add meta filter`.

Procedures concerned by access control and translation of references are: `Change references`, `Add public variables`, `Transform advice to a meta method`, `Transform parameter of advices to local variables`. Detailed discussion is given in 4.2.4.

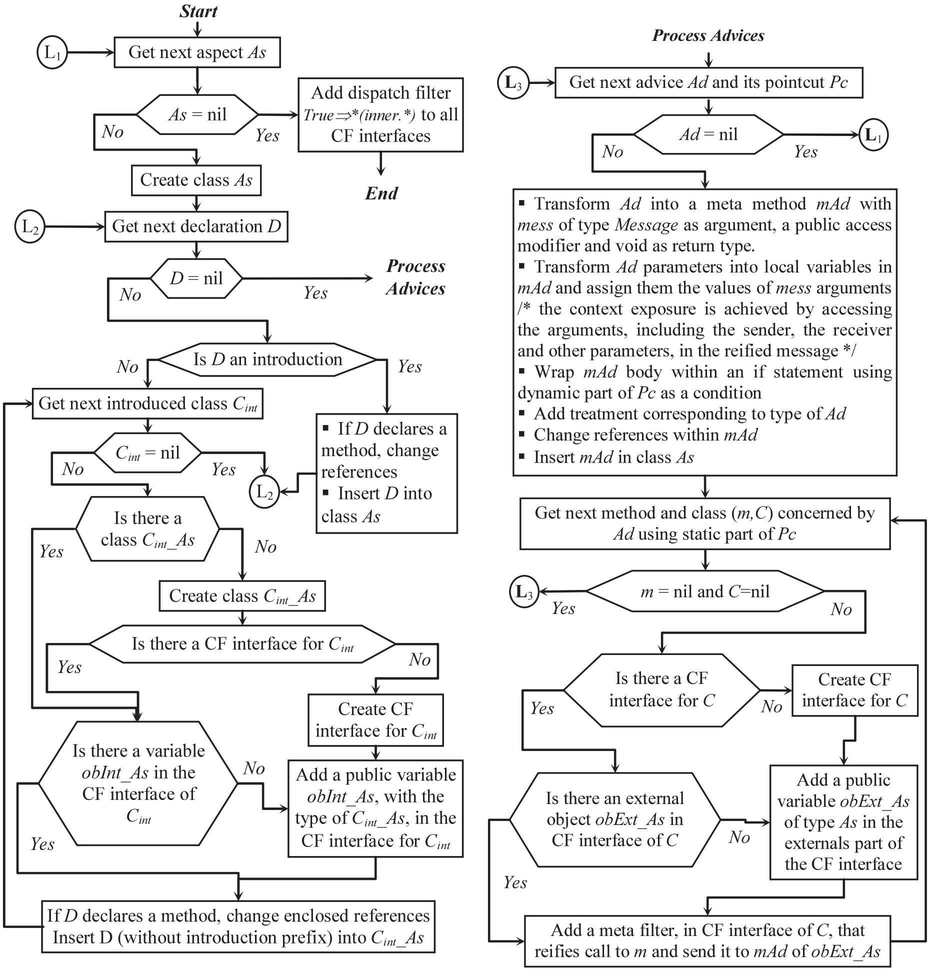
Start
L1
Get next aspect As
As = nil
No
Yes
Add dispatch filter True⇒*(inner.*) to all CF interfaces
End
Create class As
L2
Get next declaration D
D = nil
No
Yes
Process Advices
Is D an introduction
No
Yes
Get next introduced class Cint
▪ If D declares a method, change references
▪ Insert D into class As
Cint = nil
No
Yes
L2
Is there a class Cint_As
Yes
No
Create class Cint_As
Is there a CF interface for Cint
Yes
No
Create CF interface for Cint
Is there a variable obInt_As in the CF interface of Cint
No
Yes
Add a public variable obInt_As, with the type of Cint_As, in the CF interface for Cint
If D declares a method, change enclosed references
Insert D (without introduction prefix) into Cint_As
Process Advices
L3
Get next advice Ad and its pointcut Pc
Ad = nil
No
Yes
L1
▪ Transform Ad into a meta method mAd with mess of type Message as argument, a public access modifier and void as return type.
▪ Transform Ad parameters into local variables in mAd and assign them the values of mess arguments /* the context exposure is achieved by accessing the arguments, including the sender, the receiver and other parameters, in the reified message */
▪ Wrap mAd body within an if statement using dynamic part of Pc as a condition
▪ Add treatment corresponding to type of Ad
▪ Change references within mAd
▪ Insert mAd in class As
Get next method and class (m,C) concerned by Ad using static part of Pc
L3
m = nil and C=nil
Yes
No
Is there a CF interface for C
Yes
No
Create CF interface for C
Is there an external object obExt_As in CF interface of C
No
Yes
Add a public variable obExt_As of type As in the externals part of the CF interface
Add a meta filter, in CF interface of C, that reifies call to m and send it to mAd of obExt_As

Fig. 7. ASPECTJ to CF translation algorithm.

4.2.3. *Example*

To illustrate the translation algorithm, we use the example given in Listing 1. It consists of two base code classes C1 and C2, and two aspects A1 and A2. Aspect A1 declares its precedence over A2 and introduces j, getj(), setj() and test() into C1. It also declares h, seth() as local members. A1 contains 3 advices. The first specifies some treatment (body 5) to do before any call to m1(). The second advice is an around advice that may alter parameter value of the calls to m2() and returns two times whatever m2() returns. The third advice concerns both C1 and C2, it specifies some treatment (body 6) to execute after the call of m2().

Aspect A2 introduces f and m1() into C2 and s into C1. It also specifies two advices. Both have pointcuts with dynamic parts: (this(C1)||this(C2)) and cflowbelow(...).

```
public class C1 {
  public int i;
  private Boolean b;
  public void m1(String S) {
  . . . /* Body 1 */ }

  public int m2(int k) {
  . . . /* Body 2 */}
  . . .

}
  public class C2 {
  public int count;
  public void m2() {
  . . . /* Body 3 */ }
  public void m3() {
  . . . /* Body 4 */ }
}
```

```
aspect A1 {
  declare precedence A1, A2;
  private int h;
  private void seth(int v) {h = v;}
  public int C1.j;
  public int C1.getj() { return j;}
  public void C1.setj(int k) { j = k;}
  public Boolean C1.test() { return i == j;}
  ......................................
  before(String s) : call(* C1.m1(String))
  &&args(s) && target(C1)
  {. . . // Body 5 }
  ......................................
  int around(int k, C1 t) :
  call(int C1.m2(k)) && target(t) {
      if (t.test()) h = proceed();
      else h = proceed(k-2);
      t.setj (t.getj() + t.i);
      return h*2; }
  after() : call(* *.m2(..)) { . . . /* Body 6 */}
}
```

```
aspect A2 {
  public float C2.f;
  public Boolean C2.m1()
  { . . . /* Body 7*/ };
  private String C1.s;

.........................
before() : call(* C2.m3())
&& ( this(C1) || this(C2))
{
  . . . // Body 8
}
.........................
after() : call(* *.m2(..))
&&
!cflowbelow(call(**.m2(..))
{
. . . // Body 9
}
}
```

Listing 1. Example of an ASPECTJ program.

When applying the translation algorithm we get four classes `A1`, `A2`, `C1_A1`, `C2_A2` and two CF interfaces. `A1` contains declarations of `h`, `seth()` and methods `mBefore_1(), mAround_2(), mAfter_3()`. `A2` contains only `mBefore_1()` and `mAfter_2()`. C1_A1 contains members introduced by aspects `A1` and `A2`, and `C2_A2` contains only members introduced by `A2` (see Listing 2). Additional discussion of this translation is given in 4.2.4.

4.2.4. *Discussion*

In this subsection we discuss how some parts of the translation work and how ASPECTJ features and semantics can be preserved.

Enforcing Advice Precedence. When multiple pieces of advice must be applied to the same join point, deciding which one executes first is determined by the advice precedence (Kniesel, 2009). In ASPECTJ, the precedence is governed by rules that take into account the precedence declaration clause, aspect-subaspect relationship, and the declaration order of advices within the same aspect.

ASPECTJ semantics state that when two pieces of advice are defined in different aspects, then three cases are possible:

- If aspect `A` is matched earlier than aspect `B` in some `declare` precedence form, then all advice in concrete aspect `A` has precedence over all advice in concrete aspect `B` when they are on the same join point.
- Otherwise, if aspect `A` is a subaspect of aspect `B`, then all advice defined in `A` has precedence over all advice defined in `B`. So, unless otherwise specified with declare precedence, advice in a subaspect has precedence over advice in a superaspect.

```
public class A1 {
  private int h;
  private void seth(int v) {h = v;}
  public void mBefore_1(Message mess){
   private String s=
   mess.target().getArgumentAsString(1);
  // No dynamic part in the pointcut
  . . .// Body 5
  mess.continue();
}
..............................................
public void mAround_2(Message mess){
  private int k = mess.target().getArgumentAsInt(1);
  private C1 t = (C1) mess.target();
  if (t.obInt_A1.test(t)) h = mess.send();
  else { mess.putArgument(1, Message.reifyInt(k-2));
      h = mess.send();
  }
  t.obInt_A1.setj (t.obInt_A1.getj() + t.i);
  mess.reply(h*2);
}
..............................................
public void mAfter_3(Message mess){
  mess.send();
  . . .// Body 6
  mess.continue();
}
..............................................
public class C1_A1 {
  public int j;
  public int getj() { return j;}
  public void setj(int k) { j = k;}
  public Boolean test(C1 ob) { return ob.i == j;}
  public String s;
}
..............................................
public class C2_A2 {
  public float f;
  public Boolean m1() { . . . /∗ Body 7 */} }
```

```
public class A2 {
  public void mBefore_1(Message mess){
    if (DynamicPC.thisPc(mess.sender(),"C1")||
    DynamicPC.thisPc(mess.sender(),"C2")){
    . . . /*Body 8*/ }
   mess.continue();
}
..........................................
  public void mAfter_2(Message mess){
    mess.send();
    if (!DynamicPC.cFlowBelowPc(mess){ . . .
    /*Body 9*/ }
    mess.continue();
  }
..........................................
class C1 interface {
internals
  public C1_A1 obInt_A1;
  public C1_A2 obInt_A2;
externals
  public A1 obExt_A1;
inputfilters
  meta1 : meta ={ m1(obExt_A1.mBefore_1)}
  meta2 : meta ={ m2(obExt_A1.mAround_2)}
  meta3 : meta ={ m2(obExt_A1.mAfter_3)}
  meta4 : meta ={ m2(obExt_A2.mAfter_2)}
}
..........................................
class C2 interface {
internals
  public C2_A2 obInt_A2;
externals
  public A1 obExt_A1;
  public A2 obExt_A2;
inputfilters
  meta1 : meta ={ m2(obExt_A1.mAfter_3)}
  meta2 : meta ={ m3(obExt_A2.mBefore_1)}
  meta3 : meta ={ m2(obExt_A2.mAfter_2)}
}
```

Listing 2. CF program generated by the translation algorithm.

- Otherwise, if two pieces of advice are defined in two different aspects, it is undefined which one has precedence.

If the two pieces of advice are defined in the same aspect, then there are two cases:

- If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
- Otherwise, the one that appears earlier in the aspect has precedence over the one that appears later.

Our translation algorithm deals with these rules in this way:

`Get next aspect`: gets aspects according to the declare precedence statement and aspect-subaspect relationship.

`Get next advice`: gets advice that appears earlier in an aspect.

`Add treatment corresponding to an advice`: inserts `send()`, `continue()` or `reply()` in specific places reflecting the advice type. For example, in Listing 2, `mess.send()` appears before `body 6` in method `mAfter_3()`, this has the effect of continuing the reified message which passes to the next filter. If the next filter represents an `After` advice the message will be sent again. When there is no other filters the message is finally executed and instructions following `mess.send()` are executed in the inverse order which corresponds to the above after advice precedence rule.

`Add meta filter`: adds filters, in the filter set, in an order that corresponds to advice precedence of the considered advice.

Preserving Access Control. Members are referenced within field access and method invocation expressions. Position of the referencing expression and the referenced member within an ASPECTJ program determines four access forms:

① Access between aspect local members such as access of `seth()` to `h` in example of Listing 1.

② Access of advices to local members and access via context exposure to introduced members and advised class members such as, respectively, access of around advice to `h`, `t.setj()` and `t.i` in Listing 1.

③ Access between introduced members and access of introduced functions to members of the advised class such as access of `getj()` to `j` and `test()` to `i` in Listing 1.

④ Access of members belonging to an aspect to member introduced, in some class, by another aspect.

The translation algorithm splits each aspect into many internal classes and one external class. Consequently, two problems arise: how to maintain references between expressions and members? What is the access modifier used to declare a member? Whenever the expression and the referenced member are in the same class, we do not need to change references and the original access modifier is used. This is the case of ①. In the same way, since advices are translated to methods in the external class, they have access to local members without any change.

Access of advices via the context exposure to introduced members is translated to an access of the corresponding method within the external class to members of the internal classes. Since there is one internal object for each base code class object, access is done by using identifier of the base code object and its corresponding internal object. The former is obtained using the `target()` method of class `Message`, and the latter is obtained using the CF interface. In the generated program of Listing 2, `mAround_2()` uses `mess.target()` to get object identifier and `t.obInt_A1`, which is added by the CF interface of `C1`, to access the introduced members. Notice that access to members of advised classes uses only `mess.target()`. Access between introduced members does not require any change since they are in the same internal class. However, access to advised class members requires adding the advised object identifier as a parameter to each introduced method that accesses these members. The actual parameter is transmitted by advices using the `target()` function of class `Message` (see `test()` in class `C1_A1` and its use in mAround_2()). The access form cited in ④does not require any change since all internal objects are accessible using variables declared in the internals part. Finally, a member declared by an aspect preserves its original access modifier unless it is referenced from outside the internal/external class where it is declared; in this case, public access modifier is used. The algorithm doesn't support translation of privileged aspect access to private or package-protected members, nor supports the translation of pointcuts associated with private and package-protected members such as a call of a private function.

The updating of references proposed in our algorithm relies on two features of CF approach. First, internal and external variables are considered as if they were declared in the base code class. Second, the class Message allow us to consider reified messages as objects on which we can apply functions that allow access to arguments, target, selector in addition to the specific functions already used: continue(), reply() and send(). In Listing 2, we use a JAVA-like reflective methods such as getArgumentAsString(pos), getArgumentAsInt(pos), etc, to transform message primitive type arguments to values corresponding to the advice parameters. Pos gives the argument position. When the type of the argument is not primitive, we use getArgument(pos). We also use putArgument(pos, obj) to change the parameter in position pos with obj.

Determining if a Method is Concerned by an Advice. As previously suggested, pointcuts bring two types of information: static and dynamic. Static information is used by the translation algorithm to determine which class and which method is concerned by an advice. Primitive pointcuts execution(signature_pattern) or call(signature_pattern) in the heading of an advice indicate if it advises a method and of which class. In the case where signature_pattern matches several classes, we use pointcuts this(...) or target(...) to determine precisely which ones are concerned. Notice that among the frequently used combinations are call(...)&&target()&&this()... or execution()&&target()&&this().... In the two cases, target is statically used to determine which class is concerned. In the first case this() is only used as a dynamic information to allow execution of an advice only when a call is issued by the type specified by this(). In the second case, i.e., during any execution, this() and target() designate the same object. Pointcut within() plays almost the same role of this() when combined with call() and execution() except that this() includes sub-types but not nested types while within() captures nested types and ignores sub-types. Pointcut args() is used to establish a link between the arguments of an advice and those of a call. Since the CF approach deals only with messages, we have considered in the translation algorithm that advice pointcut contains a call() or an execution(). However ASPECTJ allows capturing other pointcuts which cannot be supported by the translation algorithm without altering the base code. These pointcuts are: set(), get(), adviceexecution(), initialization(), preinitialization(), staticinitialization(), handler(), and withincode(). In the same way, join points corresponding to the execution of aspects themselves are not supported.

Dynamic information comes from pointcuts `cflow()`, `cflowbelow()`, `this()`, `within()` and `if()`. This means that `this()` and `within()` may play a static and a dynamic role at the same time. In the translation algorithm, `if()` is inlined within the method corresponding to an advice, while the remaining pointcuts are inlined as a call to static methods of a class (called `DynamicPC`), we have added. `DynamicPC.cFlowBelowPc(...)` and `DynamicPC.thisPc(...)` used in Listing 2, computes the dynamic pointcuts they correspond to. Except `if()`, the four pointcuts are computed by consulting the current thread stack.

Translating Advices. Listing 2 shows how advices are translated into methods in a way that preserves their precedence. For `before` advices, the corresponding meta method first executes the advice code then resumes the reified message using `mess.continue()`. In around advice, we replace `proceed()` and `return` statements with, respectively `mess.send()` and `mess.reply()`. For `after` advices, the corresponding meta method first lets the message continue using `mess.send()`, then executes what corresponds to the advice code, and then lets the message finish using `mess.continue()`. Notice that, since `send()` is the first statement in the meta method, the message continues with the next filter of filter set before executing the advice code which gives precedence to other advices.

There can be three interpretations of after advice: after the execution of the join point completes normally (`after() returning`), after it throws an exception (`after() throwing`), or after it does either one (`after`). The translation algorithm doesn't support after-returning or after-throwing join points since CF doesn't provide a means to know which one occurred. Even if, the statement `mess.send()` in the meta method can be wrapped within a `try-catch` statement, the code whose call has been reified needs to throw or re-throw an exception so that it can be dealt with in the meta method level.

Aspect Associations. Aspect instances can be associated with the whole program, with `target` or `this` objects of a pointcut, or with control flow of a pointcut. The first case is the default. The second case is obtained using the declarations `perthis(pc)` or `pertarget(pc)` after the aspect name (`pc` is a pointcut). The third case is obtained by using one of the two declarations: `percflow(pc)` or `percflowbelow(pc)`. The algorithm given in Fig. 7, deals only with the default case. The use of `pertarget(pc)` means that there will be as many aspect instances as target objects of the pointcut `pc`. Hence there will be also distinct field values for each aspect instance. To cope with this case, rather than splitting an aspect into two classes (`C_As` and `As` in Section 4.2.1), we use only one class whose objects are internal objects associated, each one, with the target class objects. The same is true for `perthis` aspects but only when `target` and `this` objects are the same (i.e., when `pc` contains the primitive pointcut `execution(...)`). The other cases cannot be dealt with without affecting the base code.

Translating Named Pointcuts. The named pointcuts declared in advice headings can be translated into anonymous pointcuts so that only primitive ones are used. The elimination of named pointcuts is an easy task but we need to change their parameters. To understand this let us consider a base class `Point` having two integer fields (`x` and `y`) and two methods `setX(int newX)` and `setY(int newY)` to update respectively the `x` and `y` coordinates of class `Point` objects'. Now suppose that we need to capture the call to these two methods to ensure that some constraints are satisfied before setting the `x/y` coordinate. The following code will do the task.

```
Pointcut setter(Point p1, int newVal) :
   target(p1) && args(newVal) && (call(void setX(int)) ||
   call(void setY(int)));
before (Point p, int nVal): setter (p, nV) { /* Advice body */}
```

The elimination of a named pointcut can be done by two steps that must be achieved for each advice:

1. Replace the parameters in the pointcut declaration with those appearing in the advice. When considering the setter pointcut we get:
   ```
   Pointcut setter(Point p, int nV) :
      target(p) && args(nV) && (call(void setX(int)) ||
      call(void setY(int)));
   ```
2. Replace the specification of the named pointcut in the advice by the body of its declaration. The advice become:
   ```
   before (Point p, int nVal):
      target(p) && args(nV) && (call(void setX(int)) ||
      call(void setY(int))) {
      // The advice body
      }
   ```

Dealing with Superimposition. Rather than creating repeatedly a CF interface for each concerned class in the base code, the superimposition mechanism allows the declaration of a concern in one module consisting in three optional parts: the filter modules specification, the superimposition specification and the implementation of the behaviour (Bergmans and Aksit, 2004). While the filter module specifies how the calls are handled, the superimposition specification selects to which classes the filter module is added (Caro, 2001). The translation algorithm given in Fig. 6 does not deal with superimposition; however, it can be adapted as follows:

1. Rather than having one `CF` interface for each class, we split it in multiple `CF` interfaces where each corresponds to one aspect in the source program.
2. For each pair (`CFI, C`), where `CFI` is a `CF` interface (determined in the previous step) and `C` the concerned class, create a `CF` concern module with `CFI` as the filter module part and an expression selecting `C` in the superimposition specification part.
3. Merge `CF` concern modules (`CFC1, CFC2`, ...) having the same filter module part into one `CF` concern module (`CFC`). The latter has as superimposition specification part, the union of the superimposition specification parts of `CFC1, CFC2`, ...

Obviously the third step is the most difficult. It can be optimized through an appropriate naming of the constructs generated by the algorithm of Fig. 6. For instance, the external class name must not be dependent of base class. Notice that this adaptation does not allow getting a unique corresponding concern module for each aspect. We expect this goal to be very difficult to achieve and will require a total reformulation of our algorithm.

Unmapped Concepts. Due to the constraint of keeping the base code unchanged, some concepts had been left unmapped. Concerning the static crosscutting, the translation algorithm deals with the most important concept which is the introduction of members. The other concepts, not described in this article, such as `declare warning`, `declare soft`, `declare parent`, can be dealt with easily by changing the base code.

Concerning the dynamic crosscutting, we have proposed a mapping that deals with the most used concepts. Primitive pointcuts that are not supported or used are `set()`, `get()`, `adviceexecution()`, `initialization()`, `preinitialization()`, `staticinitialization()` and `handler()`. `set()` and get() can be easily transformed into call(...) by providing advised fields with getter and setter methods and replacing field access or field assignment with calls to these methods, and then intercept the calls by filters. However, this cannot be achieved without altering the base code.

The translation algorithm covers three advice types and context exposure using reification and class `Message`. It also deals with precedence by using `meta` filter and order of filters within a filter set. However, it doesn't deal with multiple precedence declaration statements that produce a cycle in the precedence graph, such as `As1` precedes `As2` that precedes `As3` ... that precedes `As1`. Finally, the translation algorithm doesn't cope with access of advices to base code class private members nor copes with join points related to `private` and `package protected` members.

5. Related Work

Because CF and ASPECTJ are relatively new paradigms and since they are continually evolving, only a limited amount of research work is devoted to their assessment and comparison. In our context, the closest related works are those concerning implementation of AOP concepts and weaving techniques (Garcia, 2003; Hilsdale and Hugunin, 2004; Wichman, 1999). In Garcia (2003), the author proposes a specific implementation of CF upon the .Net platform. More specifically, the .Net extension is composed of a runtime system for the interpretation of CF specifications that are added to new applications as well as to legacy systems. The work in Hilsdale and Hugunin (2004) is dedicated to the weaving approach used for ASPECTJ and particularly the concepts and mechanisms used to weave the advices. The work described in Wichman (1999) consists of an implementation of the CF approach called ComposeJ. The author shows how successive source code transformations, directed by the composition filter specification, produce pure Java statements.

Despite the fact that these works do not have a comparison goal, they have some similarities with the work presented in this article. Indeed, in our work, each translation proposes a specific algorithm that implements one approach using the other. Moreover, we have achieved a particular weaving of the main concepts under a challenging constraint: that of preserving base code classes unchanged.

6. Conclusion

Aspect oriented programming is a new paradigm that seeks better decomposition of software systems. Although AspectJ and CF are among the most well-known and mature AOP technology available today, there is no attempt that aims at comparing deeply the two approaches. As a step towards this goal, we presented a mapping of the two approaches. The originality of this work lies in the fact that it puts to the test the two approaches by directly confronting their concepts. Consequently, our mapping can be used as a basis for a comparative study of the two approaches as well as a tool for transforming concerns in a model integrated computing environment where coexist AspectJ and CF.

Throughout this work, it appears that even though each approach can be expressed using the other, neither subsumes the other. In all the practical software cases studied, a straightforward mapping was not possible; it seems also that achieving a one-to-one mapping is impossible given that, at least, an aspect in AspectJ may concern many classes whereas an input filters set concerns only one class.

There are several perspectives to this work. Those currently in progress are mappings between other SOC languages, the development of a metamodel transformation tool dedicated to a model integrated computing environment where AspectJ and CF are considered as two metamodels, and finalizing a comparative study of AspectJ and CF.

References

Aho, A.V. *et al.* (1986). *Compilers. Principles, Techniques and Tools*. Addison-Wesley.

Aksit, M., Tekinerdogan, B. (1998a). Aspect-oriented programming using composition filters. In: *ECOOP'98 Workshop Reader*. Springer-Verlag, pp. 435–436.

Aksit, M., Tekinerdogan, B. (1998b). Solving the modelling problems of object-oriented languages by composing multiple aspects using composition filters. In: *AOP'98*, workshop position paper. `http://trese.cs.utwente.nl/oldhtml/publications/publication_topics/aosd.htm`

Bergmans, L. (1994a). *Composing Concurrent Objects*. PhD thesis, University of Twente. `http://trese.cs.utwente.nl/oldhtml/publications/publication_topics/aosd.htm`

Bergmans, L. (1994b). *The Composition Filters Object Model*. Dept. of Computer Science, University of Twente.

Bergmans, L., Aksit, M. (2001). Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10), 51–57.

Bergmans, L., Aksit, M. (2004). Principles and design rationale of composition filters. In: Filman, R. (Ed.), *Aspect-Oriented Software Development*. Addison-Wesley. `http://trese.cs.utwente.nl/oldhtml/publications/publication_topics/aosd.htm`

Caro, P.S. (2001). *Adding Systemic Crosscutting and Superimposition to Composition Filters*. Master's thesis, University of Twente, The Netherlands.

Garcıa, C.F.N. (2003). *Compose*: A Runtime for the .Net Platform*. Master's thesis, Vrije Universiteit Brussel.

Gray, J., Gokhale, A. (2004). Concern separation in model integrated computing. In: *OMG's First Annual Model-Integrated Computing Workshop*. Arlington, VA, USA.

Heidenreich, F. *et al.* (2009). On language-independent model modularisation. In: Rashid, A., Ossher, H. (Eds.), *Transactions on AOSD VI: Special Issue on Aspects and Model-Driven Engineering*, *LNCS*, Vol. 5560. Springer-Verlag, pp. 39–82.

Hilsdale, E., Hugunin, J. (2004). Advice weaving in ASPECTJ. In: *3rd International Conference on Aspect-Oriented Software Development* (*AOSD*). ACM Digital Library.

Kiczales, G. *et al.* (1997). Aspect oriented programming. In: *Proc. of European Conference on Object-Oriented Programming* (*ECOOP*). *Lecture Notes in Computer Science*, vol. 1241, pp. 220–242.
`http://eclipse.org/AspectJ`

Kiczales, G. *et al.* (2001). An overview of ASPECTJ. In: *Proc. of European Conference on Object-Oriented Programming* (*ECOOP*). Springer-Verlag.

Kleppe, A. *et al.* (2003). *MDA Explained*: *The Model Driven Architecture Practice and Promise*. Addison-Wesley.

Kniesel, G. (2009). Detection and resolution of weaving interaction. In: Rashid, A., Ossher, H. (Eds.), *Transactions on AOSD V*, *LNCS*, Vol. 5490. Springer-Verlag, pp. 135–186.

Koopmans, P. (1995). *On the Design and Realization of the Sina Compiler*. MSc thesis, Dept. of Computer Science, University of Twente.
`http://trese.cs.utwente.nl/oldhtml/publications/publication_topics/implementation_filters.htm`

Laddad, R. (2003). ASPECTJ *in Action*: *Practical Aspect-Oriented Programming*. Manning Publications Co.

Meslati D., *et al.* (2006). From composition filters to ASPECTJ: A platform specific model transformation. *Journal of Computing and Information Technology-CIT*, 14(2), 111–131.

Wichman, J.C. (1999). *The Development of a Preprocessor to Facilitate Composition Filters in the* JAVA *Language*. MSc thesis, Dept. of Computer Science, University of Twente.
`http://trese.cs.utwente.nl/oldhtml/publications/publication_topics/implementation_filters.htm`

D. Meslati received a master degree in computer science from the University of Annaba (Algeria) and a doctorate in computer science from the same university in 2005. His research interests are software development and evolution methodologies, separation of concerns and bio-inspired software systems. Dr. Meslati, is currently an associate professor in the Department of Computer Science at the University of Annaba. He is also the head of the research group on evolution and reuse of software systems in the LRI Laboratory of the same university.

ASPECTJ ir kompozicijos filtrų sąvokų sugretinimas

Djamel MESLATI

AspectJ ir kompozicijų filtrai yra dvi pačios svarbiausios aspektinio programavimo kalbos, sukurtos per pastarąjį dešimtmetį. Nors abi šios kalbos yra palyginti brandžios ir jų tobulinimo bei praktinio panaudojimo klausimams yra skirta daug mokslinių tyrimų, kol kas nebuvo rimtesnių bandymų palyginti jas tarpusavyje. Straipsnyje žengta šia linkme. Jame pasiūlyta kaip vienos kalbos sąvokas atvaizduoti į kitos kalbos sąvokas, kas sudaro galimybes tas sąvokas sugretinti ir palyginti tarpusavyje. Straipsnyje parodyta, kad nors abi kalbos yra Java kalbos plėtiniai ir priklauso tai pačiai programavimo kalbų kategorijai, atvaizdis nėra nei griežtas, nei vienareikšmis.