CompAr: Ensuring Safe Around Advice Composition

Renaud Pawlak, Laurence Duchien, Lionel Seinturier

CompAr: Ensuring Safe Around Advice Composition

Renaud Pawlak[1], Laurence Duchien[1], and Lionel Seinturier[2]

[1] Université de Lille, INRIA Futurs-Jacquard
Bâtiment M3, Villeneuve d'Ascq, 59655, France
[2] Université de Paris 6, LIP6, INRIA Futurs-Jacquard
4, place Jussieu, 75252 Paris, France

Abstract. Advanced techniques in separation of concerns such as Aspect-Oriented Programming, help to develop more maintainable and more efficient applications by providing means for modularizing crosscutting concerns. However, conflicts may appear when several concerns need to be composed for the same application, especially when dealing with around advice. We call this problem the Aspect Composition Issue (ACI). Based on our experience in programming aspects, this paper presents a language called CompAr, which allows the programmer to abstractly define an execution domain, the advice codes, and their execution constraints. The CompAr compiler then evaluates the definitions in order to check if the execution constraints are fulfilled. Using a concrete AOP case study, we show how to use the CompAr language in order to detect and avoid ACIs.

1 Introduction

When dealing with complex software, programmers and designers naturally try to apply the *divide and conquer* principle by splitting the application into small pieces, which are easier to understand than the whole system. This technique is referred to as Separation of Concerns (SoC) and has been originally described in [12, 4]. The goal of SoC is to analyze one of the parts without having to take the other parts into account. However, in the difficult process of making the application parts independent, many issues can arise.

Within the last few years, Aspect-Oriented Programming (AOP) [8] has stressed the point that some concerns are significantly difficult to modularize. AOP identifies these concerns as crosscutting concerns, i.e. the implementation of these concerns spans over some modules of the other concerns. AOP and related approached propose some solutions to these issues which would pull out the crosscutting concerns from the application code, allowing for easier modularization.

Thanks to AOP, some techniques that are used in middleware and other fields have been highlighted and are becoming more popular. One of the most important and widely used techniques can be referred to as the *around advising of the code*; this is an important mechanism used to compose concerns together. It

has been employed under several contexts and can be implemented by wrappers, filters, interceptors, proxies, around code injections, and so on. However, all these implementation techniques face the same issues when composing several concerns. We call this kind of issue the Aspect Composition Issue (ACI) [13]. Unfortunately, little support is provided to solve this problem and, most of the time, it has to be handled manually, without any tools or guidelines.

In this paper, we present a language called CompAr that is able to automatically detect conflicting around advice codes out of an abstract specification of the aspect-oriented program.

In section 2, we define the around advice and give some examples of its use. Then, in section 3, we present our case study, which deals with real-life aspects that we use for reasoning on specific ACIs. In section 4, we define CompAr, our language to specify composition-relevant information. We show how CompAr checks a specification to detect composition issues. Section 5 goes further in studying ACI by focussing on composing all the aspects of our case study by using CompAr. Finally, before concluding, we list some related works in section 6.

2 The Around Advice

Breaking down software into independent modules, objects or components that can be designed or programmed separately, implies a composition phase. In this section, we focus on a useful composition mechanism called around advice; we introduce its mechanisms and common utilizations.

2.1 Introducing Around Advice

When composing several modules together, structural or behavioral composition mechanisms are needed; our focus is on the behavioral compositions. When composing behaviors, the behavior of the target module is modified by another behavior coming from source modules. In order to achieve this, around advice is a convenient mechanism; we describe the device here in an informal manner.

One can apply an around device code to a given target executable element such as a method, a constructor, or a field access; as a result, the target element will be modified transparently for the base program; the target element is said to be advised. Once this element is executed in the program, the flow of execution is the following:

1. the advice code is executed (and can access some contextual information from the base program),
2. when a special instruction called *proceed* is reached, the advice code executes the advised executable element (called *this*),
3. when the advised method ends, the rest of the advice code is executed and can finally return.

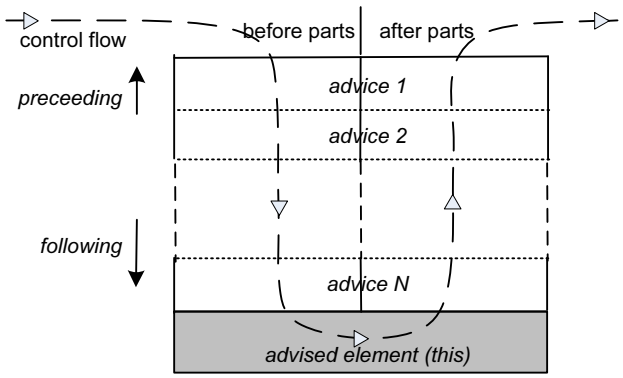

Fig. 1. An around advice chain.

Note that an advice code has a *before* part (before the call to *proceed*) and an *after* part (after the call to *proceed*). It is not required to call *proceed* (it then completely replaces the implementation of the advised method).

As shown in figure 1, a method can be advised several times; this method then holds an advice chain. Each advice before part is executed in the order defined by the chain (the top advice code is executed first). When the execution of an advice code reaches *proceed*, the control is passed to the next advice of the chain.

The advised element is executed at the end of the chain and returns so that all the advice after parts are executed in the reverse order of the before parts execution order. An advice code can break the regular advice chain control flow by not calling proceed or by throwing an exception.

Note that an advice code is said to *precede* the advice codes that come after it in the chain and to *follow* the advice codes that come before it in the chain.

2.2 Around Advice Utilizations

Even though around advice-related techniques are used in many contexts and languages, the main utilization of around advice is in Aspect-Oriented Programming (AOP) [8]. AOP focuses on solving *crosscutting* of concerns when programming or designing complex applications. In short, AOP solves crosscutting and tangling code issues by allowing the programmer to define *pointcuts*. A pointcut is a set of points (joinpoints) in the base program that are affected by aspect-level advice, including around advice.

Aside from the popular AspectJ [7] language, numerous projects have included AOP features such as JAC [14], CFOM [2], CaesarJ [10], and PROSE [15]. Besides, several other approaches use techniques that can be closely related to around advising, for instance: Composition Filters [2] and Multi-Dimensional Separation of Concerns [11].

When a new language is not available, around advice is usually implemented by using Interception [9] [14], which is a very popular mechanism to imple-

ment separation of concerns in middleware environments. In these frameworks, interceptors are regular objects which may intercept the method invocations, the object constructions, and/or the field accesses. Chains of interceptors are functionally equivalent to around advice chains.

The following code shows the implementation of an interceptor using the AOP-Alliance [1] Java interfaces (the common Public Domain interfaces that have been defined and implemented by some of the the aforementioned projects).

```
class MyInterceptor implements MethodInterceptor {
  Object invoke(MethodInvocation i) throws Exception {
    System.out.println("About to invoke...");
    return i.proceed();
} }
```

3 An AOP Case Study for Composition

This section presents a set of useful generic server-side aspects which are typically used in distributed middleware layers.

For the sake of this paper, we have focused solely on the around devices of the aspects, and have simplified them to keep only the relevant details. Note that we use the AOP Alliance API in order to remain as independent as possible from any specific language or commercial framework. The aspects depicted here are used to illustrate typical composition issues, and they will be formalized in the next sections.

3.1 Logging Aspect

The most well-known and straightforward application of AOP consists of seamlessly introducing logging when needed. By using around advice, the logging aspect can write into files what happens on a server; this can be useful for maintenance (security, performance, debugging). As shown below, the implementation of the logging aspect's around advice is quite simple.

```
class LoggingAspect implements MethodInterceptor {
  Object invoke(MethodInvocation i) throws Exception {
    Object result=null;
    logEntry(i.getMethod(),i.getParameters());
    result=i.proceed();
    logExit(i.getMethod(),result);
    return result;
  } [...] }
```

3.2 Authentication Aspect

Within a client/server interaction, a server-side authentication aspect checks that the user associated to the current session has the right to access the involved

resources. If the current session has no associated user, the authentication aspect may ask the client to authenticate by, for example, asking for a login and a password. If the client does not have the right to perform the current action, the authentication aspect performs an alternative action, such as throwing an exception to notify the client that the rights were not granted.

The implementation of the authentication aspect's logic is mainly done within the following around advice:

```
class AuthenticationAspect implements MethodInterceptor {
  Object invoke(MethodInvocation i) throws Exception {
    // gets the session from a thread local (set by a client)
    Session s=getThreadLocalAttribute("session");
    if(s.getUser()==null) doAuthentication(s);
    if(canAccess(s.getUser(),i.getMethod())) {
      return i.proceed();
    } else {
      throw new AuthenticationException(
      "user '"+s.getUser()+"' cannot access '"+i.getMethod()+"'");
    }
  } [...] }
```

3.3 Persistence Aspect

On the server, objects can be persistent. Typically, this is achieved by advising all the setters and getters of the objects and by writing or reading the data in a storage (XML files, JDBC data source). In many systems, the object's fields may still be directly accessed on optimization purpose so that the object acts as a cache for the storage. Additionally, any transient object that is referenced by a persistent object (through a reference or a collection) should itself become persistent.

The implementation of the persistence aspect's logic is mainly done within the following around advice:

```
class PersistenceAspect implements MethodInterceptor {
  // advice all the setters, getters, adders, and removers
  Object invoke(MethodInvocation i) throws Exception {
    Object result=null;
    if (isPersistent(i.getThis()) {
      [...] // Before: read needed data from storage
      result = i.proceed() // read or write the value in memory
      // After: write changed data into storage and
      if(isSetter(i.getMethod())) {
        // make the new referenced object persistent if needed
        if(isStorable(i.getParameters()[0]))
          makePersistent(i.getParameters()[0]);
      [...] } [...]
    } else result = i.proceed(); // Transient object case.
    return result;
  } [...] }
```

3.4 Association Aspect

In object or component models, entities may be related to each other through references or collections. At a higher level, these references or collections can be part of an association; they are then called *roles*.

For instance, an association exists between an employee and a company: an employee *belongs to* a company and a company *employs* several employees; each class (`Employee` and `Company`) defines a role field of this association. When a role that is part of an association is set, as through a role setter, it generally means that the other role should be updated in order to preserve the association integrity.

With AOP, it is possible to handle the association integrity concern within an aspect. This concern, which is usually a crosscutting one, can then be cleanly modularized and the maintenance of the application is more straightforward. The main logic of the association integrity concern is programmed in the following around advice:

```
class AssociationAspect implements MethodInterceptor {
  // advice the setters, adders, remover of roles
  Object invoke(MethodInvocation i) throws Exception {
    Field current = getCurrentRole(i.getMethod());
    if(current!=null) {
      // do not update if we are already within updating
      if (getThreadLocalAttribute("update") ==
          getCurrentRole(i.getMethod()))
        return i.proceed();
      Field opposite = getOppositeRole(i.getMethod());
      try {
        setThreadLocalAttribute("update", opposite);
        doUpdate(opposite, // the opposite role
          i.getThis(), // the object that holds the current role
          current.get(i.getThis()), // the old role value
          i.getParameters()[0]); // the new role value
      } finally { setThreadLocalAttribute("update", null); }
    }
    return = i.proceed();
  } [...] }
```

3.5 Composing `Logging`, `Authentication`, and `Persistence`

As a first introduction to ACIs, this section informally shows how to compose the `Logging`, `Authentication`, and `Persistence` aspects. This simple composition problem illustrates the importance of correctly ordering the around advice.

When composing these three aspects, a simple reasoning can help the programmer to find the aspect interactions and thus find out how to solve them. Let us first look at the code of these aspects (see sections 3.1, 3.2, and 3.3). A quick glance shows that the only aspect that does not call proceed all the time is the authentication aspect. This property is important because when an aspect

is programmed independently from any context, the programmer assumes that the invocation is actually proceeded to the advised element.

Therefore, if the logging around advice is placed after the authentication advice, then the logging will only be performed if the invocation is authenticated. However, it is not always the behavior that a programmer would expect for the system. Indeed, to detect attack attempts on the server, we may want to log all the requests, even if the associated action is not successfully executed. On the contrary, a quick study of the persistence aspect shows that we want to apply the persistence only if the action is successfully executed.

Finally, we are in the presence of three significantly different kinds of around advice.

- The logging has an execution constraint which states that its before part must always be executed. By using first order action logic, this constraint can be expressed as: `beforeLogging`, where `beforeLogging` is true if the before-logging part has been executed at the end of the advice chain execution. We refer to this kind of advice as **obligatory** advice.
- The persistence has an execution constraint which can be expressed as: `[persistent]?this<=>writeStorage:true`, to be read as "if our execution context is `persistent` (boolean), then the execution of `this` advised implies the execution of the `writeStorage` action and vice versa, else the execution constraint is always fulfilled (`true`)". Note that we refer to this kind of advice as **exclusive** advice.
- The authentication does not have an execution constraint, but it does not always call proceed. We refer to this kind of advice as **conditional** advice.

Taking into account all that has been said, in order to fulfill the obligatory and exclusive constraints, the best order for the chain should be: `(Logging > Authentication > Persistence)`. This is, on the other hand, a very intuitive result that would need to be validated. Besides, when the number of aspects grows, it can become tedious to manually fulfill all the execution constraints. In the next section, we present CompAr, a language which helps the programmer to find and validate the right composition order in a rigorous way.

4 Supporting Aspects Composition: the CompAr Language

In the previous section, we presented a set of useful aspects for server-side middleware layers: `Logging`, `Authentication`, `Persistence`, and `Association`. Thanks to AOP and around advising, we have been able to separately define these different concerns so that the understanding of the sever-side system is easier. However, as seen in section 3.5 some ACIs are likely to appear when composing the aspects. This is due to the fact that each aspect is programmed independently, and holds some *implicit constraints*.

In order to deal with the ACIs, we have defined a language called CompAr (for Composing Around advice). CompAr allows the programmer to specify the

advice codes and their implicit constraints. In addition, CompAr checks that a given composition order is valid for a set of execution contexts.

In the rest of this section, we present CompAr (4.1), we define its semantics (4.2), and we apply it to our composition example (4.3).

4.1 The CompAr Language

In order to introduce CompAr, we first show how to specify the composition problem which was informally presented in 3.5. In order to do this, we write the following abstract program:

```
choices: persistent, authenticated;
advice logging:loggingEnter {loggingEnter+loggingExit}
advice authentication { [authenticated]?-+-:throw NotAuthenticated }
advice persistenceSetter:[persistent]?this<=>writeStorage:true {
  [persistent]?-+(caller(persistent=true),writeStorage):-+-}
advised a { logging, authentication, persistenceSetter; }
```

Where the `choices` command defines the different boolean variables of the execution domain, `advice` defines a new abstract advice code, and `advised` defines a composition order to be tested by the compiler. When run, the compiler executes the defined advised in the domain (all the possible combinations of choices values) and checks that the advice definitions are valid. Note that choices can be initialized to `true` or `false` in order to restrain the execution domain, but they are usually left undefined, as in this case, to test all the possible executions.

An advice definition contains two parts: an optional post-execution constraint, defined after the advice name and separated from it by a colon; and a body, within curly brackets, which represents the abstract definition of the advice code.

As a result, with CompAr, the logging advice programmed in section 3.1 can be abstractly defined by:

`advice logging:loggingEnter{loggingEnter+loggingExit}`.

It means that the body is composed of a `loggingEnter` before-proceed action and a `loggingExit` after-proceed action. Besides, when the advice is included in an advised, the post-condition execution constraint ensures that the `loggingEnter` action has been executed.

The persistence advice is more complicated but follows the same principle:

`advice persistenceSetter:[persistent]?this<=>writeStorage:true {`
`[persistent]?-+(caller(persistent=true),writeStorage):-+-}`

As an execution constraint, we recognize the constraint defined in section 3.5. The body must be understood as follows: "If the execution context is `persistent` we define a body that does nothing as a before part and that executes two actions as an after part: (1) it sets the calling context to `persistent` (as an effect, the newly referred object is made persistent), (2) it executes the `writeStorage` action. If the execution context is not `persistent`, we just proceed the execution."

Note that we use `-` to indicate that the advised body code performs some action that is not relevant for composition.

4.2 The CompAr Semantics

We now give a brief overview of the CompAr semantics by using a denotational semantics. CompAr can be split into two sub-languages: the body language and the constraint language.

The Body Language Semantics: The body language is inductively defined by three denotation functions: $[\![B]\!]$*:Environment*$\rightarrow$*Environment* (body denotation function), $[\![E]\!]$*:Environment*$\rightarrow$ *Environment* (expression denotation function), and $[\![T]\!]$*:Environment*$\rightarrow$*Boolean* (boolean expression denotation function). For the sake of simplification, we use primitive functions that we informally describe.

The $[\![B]\!]$ denotation function is defined through the $[\![E]\!]$ denotation function:

1. $[\![E1\texttt{+}E2]\!](e) = [\![E2]\!](\text{proceed}([\![E1]\!](e)))$
2. $[\![E]\!](e) = [\![E]\!](e)$

where proceed*:Environment*$\rightarrow$*Environment* is the function that corresponds to the denotation function of the next advice body in the chain. For a given advised execution, an environment contains a linked code list, which corresponds to the advice order that has been defined for the advised. Moreover, when the end of the chain is reached and the advised element is executed, a *this* action is set to 'executed' in the environment.

The $[\![E]\!]$ denotation function (for expressions) is defined as follows and using $[\![B]\!]$ and $[\![T]\!]$:

1. $[\![\texttt{-}]\!](e) = e$ // skip function
2. $[\![\texttt{(}E\texttt{)}]\!](e) = [\![E]\!](e)$
3. $[\![\texttt{(}E1\texttt{,}E2\texttt{)}]\!](e) = [\![E2]\!]([\![E1]\!](e))$
4. $[\![i]\!](e) = e[\text{executed}/i]$
5. $[\![\texttt{[}T\texttt{]?}B1\texttt{:}B2]\!](e) = [\![B1]\!](e)$ if $[\![T]\!](e)$=*true*, $[\![B2]\!](e)$ otherwise
6. $[\![i\texttt{(}i1\texttt{=}T1\texttt{,...,}in\texttt{=}Tn\texttt{)}]\!](e)$
   $=\text{advised}(i,\text{new}[[\![T1]\!](e)/i1,...,[\![Tn]\!](e)/in],e[\text{invoked}/i])$
7. $[\![\texttt{caller(}i1\texttt{=}T1\texttt{,...,}in\texttt{=}Tn\texttt{)}]\!](e) = \text{parent}(e)[[\![T1]\!](e)/i1,...,[\![Tn]\!](e)/in]$

where advised*:Identifier*$\times$*Environment*$\times$*Environment*$\rightarrow$*Environment* is the function that initializes a new child environment (new*:*$\rightarrow$*Environment*) with the linked code that corresponds to the advised identified by i. It then proceeds the first linked code of the chain in the new environment. Note that 'advised' sets the i action state to 'invoked' in the parent environment. This i action state will be set to 'executed' in the parent environment by the 'proceed' function whenever the advised element is executed in the child environment.

Note that two types of environment changes can be performed: an action state change, where an action can be set to 'executed' or 'invoked', and a choice boolean value assignment. This assignment can be done in a new context, during an 'advised' invocation, or in a calling context, during a `caller` instruction.

We do not formally define the throwing of exceptions since it is easy to understand intuitively. When a exception is thrown, the program terminates its execution and the environment is returned as is.

Finally, we do not define the denotation function $[\![T]\!]$ because it is a classical boolean expression denotation function.

The Constraint Language Semantics: When a CompAr program is executed, it is important to note that some choices may be left undefined. As a consequence, the compiler creates all the possible environments in order to cover the domain and check out all the possible executions. For instance, When an invocation towards an advised is done, a new set of environmental contexts is created and the invocation is performed for all these contexts. Hence, the compiler creates an execution tree rather than a simple execution.

When the execution tree is created, the compiler inspects all the final environmental contexts (one per tree node) and checks, for each one, that the advice post-execution constraints are fulfilled. For a given environment, we define the $[\![C]\!]$ denotation function, which is used for constraint verification.

1. $[\![$`[`T`]?`$C1$`:`$C2$$]\!](e) = [\![C1]\!](e)$ if $[\![T]\!](e)$=*true*, $[\![C2]\!](e)$ otherwise
2. $[\![i]\!](e)$ = *true* if get$(i)(e)$=executed, *false* otherwise
3. $[\![$`?`$i$$]\!](e)$ = *true* if get$(i)(e)$=invoked, *false* otherwise
4. $[\![C1$`=>`$C2]\!](e) = [\![C1]\!](e) \Rightarrow [\![C2]\!](e)$
5. $[\![C1$`<=>`$C2]\!](e) = ([\![C1]\!](e) \Rightarrow [\![C2]\!](e)) \wedge ([\![C2]\!](e) \Rightarrow [\![C1]\!](e))$

... the rest is regular boolean expressions

4.3 Testing our Case Study with CompAr

Note that the CompAr language and all the examples used in this paper are available for download at [3].

If we compile the program defined in section 4.1, we can check that our informal reasoning led in section 3.5 is correct. The compiler then writes out:

```
[START] checking 'a' advised execution constraints...
   [OK] [0] {persistent=true, authenticated=true}
   [OK] [0] {persistent=true, authenticated=false}
   [OK] [0] {persistent=false, authenticated=true}
   [OK] [0] {persistent=false, authenticated=false}
[END] no composition errors found while checking 'a'
```

As we can see, the compiler checked the post-execution constraints for the domain, which is formed out of the combination of the `persistent` and `authenticated` choices possible values. Here, since the values are undefined, there are four possible executions and they all fulfill the constraints for the advised: ordered as `logging, authentication, persistenceSetter`.

Next, if we try an invalid order such as persistenceSetter, authentication, logging, the compiler reports an error for each execution that does not fulfill all the constraints:

```
[START] checking 'a' advised execution constraints...
   [OK] [0] {persistent=true, authenticated=true}

   [ERROR] [0] constraint unfulfilled in 'logging' (loggingEnter)
    - final context:
        choices: {persistent=true, authenticated=false}
        actions: {}
    - execution trace:
        test([persistent]?(-+caller([persistent=true]),writeStorage)
                          :(-+-)=>true)
        enter(a.persistenceSetter=>before)
        test([authenticated]?(-+-):(throw NotAuthenticated)=>false)
      * throw(throw NotAuthenticated)

   [OK] [0] {persistent=false, authenticated=true}

   [ERROR] [0] constraint unfulfilled in 'logging' (loggingEnter)
   [...]

[END] 2 composition error(s) found while checking 'a'
```

Here, we see that the `loggingEnter` constraint defined by the logging advice is not fulfilled when `authenticated=true`.

5 Using CompAr to Solve Complex ACIs

In this section, we finalize our case study which had started in sections 3.5 and 4.3 by adding the association aspect of section 3.4. As we will see next, adding this final aspect induces a difficult ACI that we manage to detect and solve with CompAr.

5.1 Composing the Association

For the association, the advice code, applied on a role setter, is active only if the advised method is a `role` (part of an association), and if we are not already in an `update` process. Further, in the case that the `role&&!update` condition is fulfilled, the association is exclusive – the opposite role of an association has to be updated only if the current role itself is updated. As a consequence, a possibly valid order for the `roleSetter` advice can be defined in the `total` advised: `advised total { logging, authentication, persistenceSetter, roleSetter; }`.

More precisely, by using CompAr, the association around advice can be abstractly specified as:

```
advice roleSetter : [role && !update]?this <=> total:true {
    ([role && !update]?total(update=true):-) + -
}
```

where `total` is the name of the advised that defines the full order of our four aspects. The association specification should be read as follows: "the before part does nothing except proceeding; the after part invokes recursively the advised `total` if we are in a role and not in an updating process (`role && !update` condition)".

Note that this definition makes the total `advised` definition recursive. The infinite recursion is avoided by the `update=true` assignment which restricts the domain of the `total` invocation and prevents having to re-apply the `roleSetter` advice.

However, the `total` ordering leads to a conflict that we explain in the next section.

5.2 The Persistence and Association Conflict

Let us imagine that we want to apply our aspects to two objects `o1` and `o2`, where `o1` and `o2` can be linked through an association. This association has two roles `r1` and `r2`. A method `setR2` can be called on `o1` in order to set the association's role and a method `setR1` can be called on `o2` in order to set the association's opposite role. Let us also assume our initial conditions imply that `o2` is persistent, that `o1` is not persistent, and that `o1.r2` and `o2.r1` are `null`.

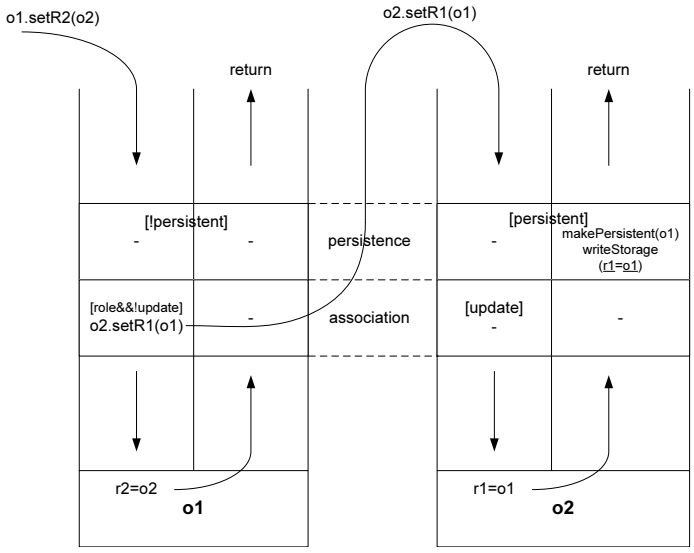

Fig. 2. Example of conflict between persistence and association.

Figure 2 shows the execution flow when `setR2(o2)` is invoked on `o1` and when the advice chain is the one suggested by the `total` advised of the previous section. Note that `x` refers to a memory variable, whereas $\underline{x}$ refers to the corresponding variable in the persistent storage.

As seen in the figure, the composition of the aspects as they are produces a side-effect that breaks the implicit persistence constraint; the final storage state (o1.r2=null) differs from the final memory state (o1.r2=o2). This composition error is mainly a result of the condition [persistent] being global to the before and after parts of the persistent advice code. As a consequence, the persistence after code, which is supposed to write the value of the r2 role in the storage, is never executed.

5.3 Solving the Conflict by Using CompAr

In the previous section, we have seen that the persistence and the association conflict. Detecting this conflict requires a great deal of analysis and understanding from the aspect designer. However, by using CompAr, this conflict can be automatically detected. In fact, if we run CompAr on the total advised as defined in section 5.1, it gives the following output:

```
[START] checking 'total' advised execution constraints...
   [OK] [0] {persistent=true, update=true, authenticated=true, role=true}
   [OK] [0] {persistent=true, update=true, authenticated=true, role=false}
   [OK] [0] {persistent=true, update=true, authenticated=false, role=true}
   [...] // checks the rest of the domain...

   [ERROR] [0] constraint unfulfilled in 'persistenceSetter'
               ([persistent]?(this<=>writeStorage):(true))
    - initial context:
        choices: {persistent=false, update=false, authenticated=true, role=true}
        actions: {}
    - final context:
        choices: {persistent=true, update=false, authenticated=true, role=true}
        actions: {loggingEnter=EXECUTED, total=EXECUTED, loggingExit=EXECUTED, this=EXECUTED}
    - execution trace:
            enter(total.logging=>before)
          * execute(loggingEnter)
            test([authenticated]?(-+-):(throw NotAuthenticated)=>true)
            enter(total.authentication=>before)
            test([persistent]?(-+caller([persistent=true]),writeStorage):(-+-)=>false)
            enter(total.persistenceSetter=>before)
            enter(total.roleSetter=>before)
            test([role&&!update]?(total(update=true)):(-)=>true)
          * invoke(total(update=true))
          * execute(total=>this)
            enter(total.roleSetter=>after)
            enter(total.persistenceSetter=>after)
            enter(total.authentication=>after)
            enter(total.logging=>after)
          * execute(loggingExit)

   [OK] [0] {persistent=false, update=false, authenticated=true, role=false}
   [OK] [0] {persistent=false, update=false, authenticated=false, role=true}
   [OK] [0] {persistent=false, update=false, authenticated=false, role=false}
   [OK] [1] {persistent=true, update=true, authenticated=true, role=true}
   [OK] [1] {persistent=true, update=true, authenticated=true, role=false}
   [OK] [1] {persistent=true, update=true, authenticated=false, role=true}
   [...] // checks the rest of the domain for recursion level 1...

[END] 1 composition error(s) found while checking 'total'
```

Therefore, by simply analyzing the compiler's output, the designer can deduce that the persistence execution constraint is not fulfilled because the action

`writeStorage` is not executed at level 0 of the recursive evaluation (the final context contains a list of the executed actions). A simple solution to solve this conflict is to decouple the `[persistent]` condition. In fact, by specifying the persistence body as, `([persistent]?-:-)+([persistent]?(caller(persistent=true),writeStorage):-)`, the compiler does not report any more errors. It is then easy for the designer to report this design change in the persistence implementation of section 3.3.

6 Related works

Some interesting studies on aspect interaction are conducted in [5]. This work, which is based on a more precise definition of the AOP's pointcuts semantics, allows their authors to automatically detect the points where a potential conflict may occur (the points where several advice codes are applied). However, it does not give solutions for ordering the advice codes. We believe that this work and our work are complementary.

Works related to this article can also be found in [6] and [16]. Their authors present interesting formal design languages to specify superimpositions at an architectural level and to compose them.

7 Conclusion

In this paper, we study the Aspect Composition Issues (ACIs) when using the around advice construct, which is a significant construct for separating concerns, especially for AOP and related approaches. Our work defines a language called CompAr that allows the specification of composition-relevant information that includes boolean choices (forming the execution domain), action executions or invocations, and post-execution constraints. Our compiler then evaluates the specification within the defined domain and checks that all the execution paths fulfill the constraints.

Our study of the four real-life aspects (logging, authentication, persistence, and association) shows that our approach helps to detect and solve ACIs (see the persistence/association ACI of section 5). Besides, the fact that we define a new language makes the approach independent from existing concrete environments or languages. CompAr can then be used as a complementary tool or a DSL for helping the designers.

Finally, even though CompAr is a research prototype, our study is a proof of concept that validation of AO programs is possible. For instance, it would be possible, for a tool or language editor, to generate the abstract CompAr specification out of a real program. One could argue that we could face a state explosion problem when executing the specification. However, since the abstract specification focuses only on composition-relevant information, and that the number of around advice codes it somehow restrained in real systems, we think that this method is applicable in most cases.

We are currently working on several improvements of the language. For instance, we would like to introduce missing constructs such as exception catches, that have not been implemented yet. We also would like to enhance the post execution constraint sub-language to allow TLA-like expressions. This would allow the designer to specify advice where the performed actions must be executed in a certain order. For instance, a security aspect should always execute the `crypt` action before the `decrypt` action.

References

1. AOP-Alliance. `http://aopalliance.sf.net`.
2. L. Bergmans, M. Aksit, and B. Tekinerdogan. *Software Architectures and Component Technology*, chapter Aspect Composition Using Composition Filters, pages 357–382. Kluwer Academic Publishers, 2001.
3. CompAr. `http://www.lifl.fr/~pawlak/compar`.
4. E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
5. R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd Int. Conf. on Aspect-Oriented Software Development (AOSD'04)*, Mar. 2004.
6. M. Katara and S. Katz. Architectural views of aspects. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 1–10. ACM Press, 2003.
7. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
8. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.
9. R. Lämmel. A semantical approach to method-call interception. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55. ACM Press, 2002.
10. M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, 2003.
11. H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.
12. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
13. R. Pawlak, L. Duchien, and G. Florin. An automatic aspect weaver with a reflective programming language. In *Proceedings of Reflection'99*, July 1999.
14. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *Proceedings of Reflection 2001*, LNCS 2192, pages 1–21, May 2001.
15. A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: Efficient dynamic weaving for java. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 100–109. ACM Press, 2003.
16. J. van Gurp, R. Smedinga, and J. Bosch. Architectural design support for composition and superimposition. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, volume 9, page 287, 2002.