Detection of interferences in aspect-oriented programs using executable assertions

Jimmy Lauret1,2, Helene Waeselynck1,2, Jean-Charles Fabre1,2
1CNRS, LAAS, 7 av. du Colonel Roche, F-31400 Toulouse, France
2Univ de Toulouse, INPT, LAAS, F-31400 Toulouse, France
jlauret@laas.fr, Jean-Charles.Fabre@laas.fr, Helene.Waeselynck@laas.fr

Abstract—Aspect-oriented programming (AOP) is a technique that promotes separation of concerns. Unfortunately, it still suffers from well-known composition issues, in particular from undesirable interferences when multiple concerns are applied at the same join point. In this paper we propose an approach to detect interferences side effect using executable assertions. The assertions are inserted in the aspect chain to detect various types of interferences. The implementation is based on the AIRIA resolver construct, recently introduced to better control conflicting aspects in AspectJ. Resolvers add observation points that were lacking in AspectJ. We propose to take advantage of this to implement automated detection of interferences at execution time. We study the feasibility of this approach and demonstrate it on artificial examples.

Keywords—Aspect interference; executable assertions; verification

I. INTRODUCTION

Aspect-Oriented Programming (AOP) provides separation of concerns between application code and non-functional mechanisms. Ideally, the developers of aspects should only be concerned about their own modules. The main hypothesis in AOP is that an entity (aspect) does not need to be aware of the existence of other entities.

Unfortunately, undesirable interferences may occur when several aspects are woven at the same join point of the base code [1], [2]. For example, one aspect can prevent the execution of another one, or write a base variable that the other one reads. Since aspects are developed independently, their actions on the program context are usually uncoordinated. Local assumptions made about the behavior of one aspect can be violated after composition with the other aspects, yielding an erroneous global behavior. It is thus important to detect bugs emerging from the composition of aspects.

We propose an approach to address interferences in AOP-implemented application that is two-fold:

(i) prevention: we use an extension of AspectJ [3], namely the AIRIA resolver construct [4], to control the ordering of aspect advices;

(ii) detection: we rely on executable assertions targeting interference properties to detect undesirable behaviors.

In this paper we focus on detection for debugging purposes. The executable assertions, intended for the testing phases, check whether the implemented ordering fulfills its intended purpose, i.e., the identified properties are never violated. As the detection of undesirable interferences is a crosscutting concern, it should be encapsulated into aspects performing the checks. A feasibility issue is that AspectJ offers a limited observability and controllability over the execution of multiple aspects, weaved at a shared joint point [5]. This is why our work investigates an instrumentation solution that is based on the AIRIA AspectJ extension.

Section II introduces and exemplifies aspect interference problems. Section III presents the general approach targeted by our work. It shows that the approach is not easily feasible in native AspectJ and motivates investigation using the resolver construct (Section IV). Section V defines implementation solutions for the detection of various types of interference. We prototype the approach in Section VI. Related work and the conclusion can be found in Sections VII and VIII, respectively.

II. PROBLEM STATEMENT

A. The Interference problem

How to formalize non-interference properties has been investigated by others, using linear temporal logic [6] or a dedicated language [1]. Interferences are always defined by considering the sequential execution of aspects woven before, after or around a target instruction in the base code (the common join point). During this sequential execution, side effects occur due to read/write access to shared data (data-flow interference) or due to actions affecting the passing of control to the next advice or to the base code (control-flow interference). For example, the authors of [6] consider four cases of interference, the first two being data-flow interferences and the other two control-flow ones:

- **Change Before (CB).** Aspect A accesses a variable v of the base code, the value of which was changed by other aspects executed before A. This is an interference, because A’s behavior might differ from the one we would get if the variable v had kept its original value.
- **Change After (CA).** Aspect A accesses a variable of the base code, the value of which is later changed by other aspects executed after A. Due to the new value of the variable, A’s behavior may turn out to be inadequate, or may be partly cancelled.
• **Invalidation Before (IB).** The aspects executed before $A$ bring the system to a state which is no longer a join point for $A$, preventing thus $A$ from executing.

• **Invalidation After (IA).** The aspects executed after $A$ bring the system to a state which is no longer a join point for $A$, hence they remove a join point of $A$ after $A$ has been executed at it.

In this paper, we stick to these four cases. They are sufficient for illustrating the general characteristics of control and data-flow interferences.

It is worth noting that an interference is not necessarily a problem. It depends on the intended behavior of aspects, i.e. the expected behavior for the user. For example, we may judge that $A$'s purpose is violated if $A$ can be cancelled (IB interference case). We may then augment $A$'s specification by an explicit statement that $A$'s execution is mandatory: it eventually occurs after any arrival at a join point for $A$. Conversely, for another aspect $B$, it may be acceptable that previously executed aspects put the system in a state no longer requiring the execution of $B$.

**B. Interference example**

Like in [6], we consider three aspects: $ALog$ (logging aspect), $ACrypt$ (encryption/decryption) and $AAuth$ (authentication). Common join points are reached whenever the base code attempts to send a message. We assume that the three aspects involve before advices, that is, pieces of code to be woven before the join point. Upon arrival at a $Send$ instruction in the base code, control is passed to the aspects. They are executed sequentially, according to a certain order, e.g., $ACrypt < ALog < AAuth < Send$. Each time an aspect terminates its execution, the pointcut application conditions of the remaining aspects are re-evaluated. This determines whether these aspects are still applicable.

We now discuss various possibilities to exemplify the four cases of interference impacting $ALog$.

$ALog$ may log either ciphered messages (i.e. $ACrypt < ALog$) or clear ones (i.e. $ALog < ACrypt$). In the first case, $ACrypt$ makes a CB interference on the content of the message. In the second case, $ACrypt$ comes after $ALog$ so we have a CA interference. Depending on our expectations for $ALog$, we may forbid either one case or the other.

Control-flow interferences are illustrated by the ordering of $ALog$ and $AAuth$. Suppose $ALog$ is executed first. If authentication fails, we have an IA interference: a sending of message is logged, but the actual sending is eventually not performed. This is acceptable if the aim of $ALog$ is to record all send attempts; this is not acceptable if the aim is to record only the sent messages. In the latter case, it is preferable to execute $AAuth$ before $ALog$, which induces an IB interference (the execution of $ALog$ is cancelled whenever authentication fails).

**III. PROPOSED APPROACH**

AspectJ offers a declare precedence statement for the ordering of aspects (see Listing 1). If no precedence is declared the compiler may choose an arbitrary ordering. Precedence works at the granularity of aspects, not advices, which may be too coarse-grained. For example, at the granularity of aspects, it is not possible to define separate precedence policies for the encryption and decryption advices of $ACrypt$. Also, the policy is determined for all shared join points, irrespective of whether the join point is a sending or a receiving statement. Finer-grained resolution of conflict can be found in AspectJ extensions, like the AIRIA extension [4] used in our work. It offers a Resolver construct to define precedence policies for advices. As will be explained, we found it convenient not only for mastering the ordering of advices, but also for instrumentation purposes.

As undesirable interferences may induce subtle failures, we would like to reveal them by means of assertions. To keep separation of concerns, the assertions should be implemented by aspects that monitor the execution of other aspects.

Figure 1 exemplifies the lifecycle of an around advice $A_i$, where several advices are attached to a join point $jp_i$. We distinguish $A_i$ from other advice, because it is of interest for a non-interference property (e.g., we want to forbid a CB interference on a data read by $A_i$). Transition $\alpha$ represents the passing of control from the base code to the first advice, $\beta$ is the activation of the before part of the distinguished advice, etc. After the execution of the join point, conflicting advices are popped in the reverse order of precedence ($\delta^i, ..., \delta$). At any time, the control flow may get out of the chain of advices ($\phi$ transitions).

All these transitions must be reified for the instrumentation of the composition. For example, to observe a CB, we need to record a data value at $\alpha$ and detect its change at $\beta$. Unfortunately, our previous work [5] shows that AspectJ does not provide the required observation points. The join point model does not expose transfers of control between the base level and the aspects, like $\alpha$ and $\beta$.

This paper first shows that all transitions of Figure 1 become observable when AIRIA’s Resolvers are used to control composition issues. It then proposes an automated instrumentation approach for the observation of interferences. The expected benefits are:

• Using AIRIA, we forbid situations where the compiler would choose an arbitrary order for the advices.

• The implemented control is documented by means of assertions, making undesirable interferences explicit.

• During integration and system testing, assertions are used to check the correctness of the control.

Listing 1. Aspect precedence declaration

```java
public aspect AspectPrecedence {
  declare precedence : ACrypt, ALog, AAuth;
}
```
Assertions should facilitate maintenance and regression testing when aspects are added, removed or changed.

IV. COMPOSING ADVICES WITH ADVICES IN AIRIA

A resolver is a kind of around advice used to control the composition of aspects at shared join points, i.e. an advice for composing advices. An example is given in listing 2.

A resolver is defined through a keyword followed by a resolver name and a parameter list (sender and an empty list in Listing 2, Line 2). Then the resolver specifies when it controls the composition (through the and/or clause) and how the composition is handled (through the proceed clause). The and/or clause of a resolver works as a pointcut, except that it specifies a list of potentially conflicting advices (Listing 2, Line 3). When advices specified in the and/or clause are weaved at a common join point, the resolver is weaved at this common join point first, in order to control the execution of these advices. In other words, a resolver is an advice with higher precedence that the handled advices. Being an advice, a resolver can be handled by another resolver, yielding resolvers of resolvers.

![Diagram of lifecycle of an around advice](image)

Listing 2. Using resolver in our example

```java
aspect LogEncryptAuth {
    void resolver sender():
and(ALog.logMsg, ACrypt.encrypt, AAuth.auth){
    [AAuth.auth, ALog.logMsg, ACrypt.encrypt].
    proceed();}
}
```

It is important to note that all advices in AIRIA are around advices. Thus to implement a before advice, we just use an around advice with an empty after part. Similarly, we implement an after advice by an around advice with an empty before part. These advices are the same as in AspectJ except that they have a unique name. For instance, in Listing 2, Line 3 refers to logMsg advice within ALog aspect.

A resolver specifies fine-grained precedence between advices. In Listing 2, the resolver sender is applied at each join point where the advices ALog.logMsg, ACrypt.encrypt, AAuth.auth are weaved (as specified in the and clause). The list between brackets determines the order of execution for the proceed clause (Listing 2, line 4). In the example, AAuth.auth is applied first, then ALog.logMsg is applied and finally ACrypt.encrypt encrypts the message before sending.

The list between brackets can be seen as representing the execution chain of advices. This is convenient to us, because all transitions in the lifecycle are now made explicit. We can insert additional advices in the schedule to capture state variables, check modifications of their value, capture events, set flags, etc. Note however that Listing 2 represents a simple case where all advices are managed by one resolver. In the general case, the ordering may be defined using resolvers of resolvers, as exemplified by Listings 3 and 4. In this example, a resolver for logging and encryption (Listing 4) is proceeded by a root resolver (Listing 3).

The resulting order for the three advices is the same as in Listing 2, although this may not be intuitive. Indeed, the root resolver introduces the following constraints: LogCrypt.logCrypt < AAuth.auth < ALog.logMsg. The other resolver adds: LogCrypt.logCrypt < ALog.logMsg < ACrypt.encrypt. The global order satisfying all constraints is authentication, logging and then encryption, as in Listing 2.

The AIRIA compiler checks that, whatever the join point, a unique root resolver manages conflicts at this join point. A total execution order must be obtained from the tree of resolvers starting from the root (see [4] for details). Our insertion of instrumentation advices must cope with such arborescent structures.

Listing 3. Using resolvers of resolvers (part 1: root resolver)

```java
aspect LogEncryptAuth{
    void resolver sender():
and(LogCrypt.logCrypt, AAuth.auth, ALog.logMsg) {
    [LogCrypt.logCrypt, AAuth.auth, ALog.logMsg].
    proceed();}
}
```

Listing 4. Using resolvers of resolvers (part 2: LogCrypt)

```java
aspect LogEncryptAuth{
    void resolver sender():
and(LogCrypt.logCrypt, AAuth.auth, ALog.logMsg) {
    [LogCrypt.logCrypt, AAuth.auth, ALog.logMsg].
    proceed();}
}
```
V. CHECKING FOR INTERFERENCES

Any aspect composition issue is addressed by resolvers, with possibly resolvers of resolvers. A tree of resolvers induces a total ordering of the proceeded entities, which can be determined at compile time.

The principle of the instrumentation is as follows. An assertion is decomposed into a set of monitoring actions to be executed at some transitions of interest. The monitoring actions are provided by advices, adding to the conflicting advices already handled by the tree of resolvers. We thus need to identify placeholders in the code of resolvers (in particular, in the proceed clauses) so that if a monitoring advice is inserted at this place, then the monitoring logic will be executed at the desired transition. The approach should work for arbitrary trees of resolvers.

We first identify placeholders for exposing the transitions of Figure 1. We then present the monitoring advices to be executed at proper transitions for detecting data- and control-flow interferences.

A. Placeholders for exposing transitions at execution time

We use the previous example with resolvers of resolvers (Listings 3 and 4) to illustrate placeholders, yielding Listings 5 and 6. Symbols \( \ell_a, \ell_{\beta}, \ldots \) denote locations for inserting monitoring advices that expose the desired transitions. More precisely, each placeholder allows a pair of before/after transitions to be exposed. For example, an around advice inserted at \( \ell_a \) exposes both \( \alpha \) and \( \alpha' \): its before part is executed at \( \alpha \) and its after part at \( \alpha' \). Additional code is also needed to detect \( \phi \) transitions, when an advice raises an exception or does not perform proceed (see Listing 5, Lines 7 and 9). As the control flow gets out of the chain of advices, \( \phi \) cannot be exposed by inserted advices. Inserted advices can however set flags to indicate whether some point of interest is reached. We then expose \( \phi \) at the level of the root resolver, by code inspecting the flags.

Listing 5. Instrumentation of the root resolver

```java
aspect LogEncryptAuth {
void resolver logEncrypt() {
and(ALog.logMsg, ACrypt.encrypt){
[ALog.logMsg, ACrypt.encrypt].proceed();}}
}

throws RuntimeException
try {[\( \ell_a \), LogEncryptAuth.logEncrypt, AAuth auth, ALog.logMsg] \( \ell_{\beta}, \ell_{\gamma}, \ell_{\delta} \)}
catch (Exception e) {
// code to expose a \( \phi \) transition due to an exception raised
}
// code to expose a \( \phi \) transition due to not calling a proceed
}
```

Listing 6. Instrumentation of the auxiliary resolver

```java
aspect LogEncryptAuth {
void resolver logEncrypt() {
and(\( \ell_{\beta}, \ell_{\gamma}, \ell_{\delta} \), ALog.logMsg, ACrypt.encrypt){
[ALog.logMsg, ACrypt.encrypt, \( \ell_{\delta} \)].
proceed();}}
```

As explained above, the passing of control from the base code to the first advice is exposed by inserting a monitoring advice at location \( \ell_a \) of the root resolver (Listing 5, Line 5). It takes precedence over all the handled entities. Its before part is executed first and its after part last.

The beginning and end of an advice \( A_i \) can be exposed by inserting advices immediately before and after it, at all locations where it is proceeded in the tree of resolvers. Listings 5 and 6 show locations \( \ell_{\beta}, \ell_{\gamma} \) for the distinguished advice \( ALog.logMsg \). According to AIRIA’s ordering algorithm, the inserted monitoring advices satisfy: \( \ldots < A_{i-1} < \ell_{\beta} < A_i < \ell_{\gamma} < A_{i+1} < \ldots \). By putting logic into the before part of these advices, we wrap the execution of the before part of \( A_i \) (transitions \( \beta \) and \( \gamma \)). Likewise, we may put logic into their after part and expose the execution of the after part of \( A_i \) (transitions \( \gamma' \) and \( \beta' \)).

Finally, transitions \( \delta \) and \( \delta' \) are exposed by a monitoring advice having the lowest precedence in the chain: its before part is the last one to execute before the join point, its after part is the first after the join point. This advice must be inserted in the proceed clauses of all resolvers of the tree (see locations \( \ell_{\delta} \) in Listings 5 and 6), including resolvers that do not handle \( A_i \). In this way, we ensure that all proceeded entities of the tree have precedence over the advice at \( \ell_{\delta} \).

We have just explained how to expose the transitions in an execution chain of advices. By placing the appropriate logic at the appropriate locations, we are now able to implement the detection of interferences.

B. Checking for data-flow and control flow interferences

For space constraints, only the data-flow interferences are discussed. The detection of control flow interferences is sketched.

1) Detection of data-flow interferences:

The detection of both CB and CA consists of the use of two aspects. AStorer provides a monitoring advice that stores the values of selected variables at some point of the aspect chain. AChecker checks that the values are unchanged at some later point of the chain. In this section, for the sake of simplicity, we focus discussion on the before part of advices. It is straightforward to apply a similar approach to the verification of properties attached to the after part only (e.g., the values at transition \( \gamma' \) should be the same as at \( \delta' \)), or even to properties spanning the before and after parts (e.g., the values at \( \gamma' \) should be the same as at \( \alpha \)).

Let us assume that a CB detection is attached to a distinguished before advice \( A_i \), having an empty after part. The
corresponding monitoring advices are also before advices and select the same join points as $A_i$. In an arbitrary tree of resolvers handling $A_i$, $AStorer.store$ is inserted at $\ell_\alpha$ and $AChecker.check$ at $\ell_\beta$. The induced execution chain (Figure 2) has the store logic executed at transition $\alpha$ and the check logic at $\beta$.

Aspect $AStorer$ owns a variablesToStore data structure, specifying the variables $(v_1, ..., v_n)$ selected for CB detection. The data consists of a set of pairs (join point description, variables). The join point description is the identifier (technically, the full qualified name) of the join point, and variables are state variables of this join point. $AStorer$ parses variablesToStore to know the variables to store. Then, through Java Reflection API\(^2\), the join point structure is accessed to retrieve the current value of the variables. The values are stored in a data structure called storedVariables, shared by the $AStorer$ and $AChecker$ aspects.

The advice $AChecker.check$ is inserted just before the execution of Target advice. The list of variables to check are retrieved from its data structure variablesToCheck. Their values are read via the join point structure. Finally, $AChecker$ compares these values to the ones in storedVariables. If they are different, an interference is reported.

CA detection follows the same principle with the storer at $\gamma$ and the checker at $\delta$.

2) Detection of control flow interferences:

The detection of IB and IA uses flags to represent the occurrence of expected events in the chain of advices. The initial value of a flag indicates that the event has not happened yet. When the event occurs, the flag value is changed. At the end of the advice chain, the root resolver is able determine whether an expected event happened or not.

For example, IB Detection requires us to verify whether Target Advice is always executed after transition $\alpha$. A first advice initializes the flag at $\alpha$ and another one sets the flag at $\beta$. Pieces of code placed in the root resolver expose a $\phi_1$ transition if the flag has not been set. IA detection follows a similar principle.

VI. PROOF OF CONCEPT

In order to prototype the proposed instrumentation, we have realized a series of experiments. These experiments include the use of multiple assertions that monitor interferences in the before and after parts. To demonstrate the correct identification of the placeholders, all experiments use a random tree structure generator. Our generation function starts from a base case where a single resolver manages the whole chain of advices (tree of depth 1). Then, it randomly extends the tree structure, adding levels of resolvers in such a way that the base case order of advices is preserved. In each case, we verified that the code of resolvers - automatically produced from a tree description - is compiled without errors and induces the intended execution order.

This series of experiments checks that:

- We are able to detect interferences in different parts of an advice chain (before and after parts).
- Monitoring advices are free of side effect, i.e. assertions targeting different properties can be safely composed.

The experiments involve an artificial example with four conflicting advices. $N_1$, $N_2$ are neutral advices performing read-only accesses to a base variable $v$; $W$ writes $v$; $Ab$ may abort the execution chain of advices. The artificial example is actually a generalization of the previous example where the logging advice was neutral, encryption exemplified a writer and authentication could abort execution. Read/write accesses are performed in both the before and after parts of advices, and $Ab$ comes in 4 versions: (1) no abort, (2) exception throwing in the before part, (3) proceed not called, (4) exception throwing in the after part. The fact that $N_1$ and $N_2$ read the same variable $v$ yields cases where several monitors of $v$ exist, that must be kept independent.

An experiment is characterized by three input parameters:

- A tree structure of resolvers, determining the order of execution of the 4 advices (among 24 possible orders).
- A set of non-interference properties to check. We have up to 16 properties, corresponding to the CA, CB, IA, and IB interferences that $W$ and $Ab$ make on the neutral advices, before or after the execution of the join point.
- The selected version of $Ab$ (among 4 possible versions).

We generated a sample of 14,400 parameter configurations. The tree structures cover all possible orders. The generated sets of properties include the maximal case with all properties to check. All experiments were successful.

VII. RELATED WORK

Our work can be compared with two categories of work:

- work targeting the composition of multiple aspects, yielding alternatives to AIRIA.
- work targeting the on-line observation of properties related to the execution of aspects.

Falling in the first category, the OARTA approach [7] defines a domain-specific language for the composition of advices in AspectJ. The execution chain of advices is not made visible, which would be a problem for our implementation of assertions. The Reflex AOP kernel [8] provides an extensible API to control the composition of aspects. It is possible implement composition in such a way that the execution chain of advices is made explicit. Then, it should be possible as well to insert instrumentation advices, similarly to what we proposed in the framework of AIRIA. The POPART runtime [9] allows for dynamic aspect composition strategies tailored to the current execution context. The underlying language is a Meta-Object Protocol based language called Groovy. It provides the observability required to detect

\(^2\)http://java.sun.com/developer/technicalArticles/ALT/Reflection/
aspect interferences [5]. However, it lacks the powerful pointcut descriptors that AspectJ offers. As regards aspect composition, AIRIA does not have the dynamic facilities of POPART, but we had the possibility to check for a total precedence order at compile time.

The second category of work is on observing the execution of aspects. Falling in this category, the AdviceTracer tool [10] attaches assertions to test cases, like the number of times an advice should be activated during the execution of the test case. Such assertions are checkable using the AspectJ join point model, while the non-interference properties we consider are not.

Closer to our work, the detection of data-flow interferences has been investigated in [11], in the framework of a Smalltalk extension. The approach consists in tagging advices with Compositional intentions.

For example, a tag may indicate that the target advice writes a certain variable, and reads another variable that should not have been changed by other advices. A CB detection then proceeds by interpreting execution as a sequence of tags, and by looking for inconsistencies (i.e., a write tag followed by a read unchanged tag). Verification is at the tag level: the tags must correctly abstract the execution of advices. In contrast, our approach checks the actual execution of advices. Moreover, we consider control-flow interferences that are not covered by [11].

VIII. CONCLUSION

To avoid undesirable interferences, controlling the order of conflicting advices is mandatory. In addition, we propose to instrument the code with executable assertions, attaching non-interference requirements to the composition of advices.

Instantiating this approach in the framework of AspectJ raises feasibility problems: AspectJ does not provide all the observation points required to implement assertions. We thus had to consider a solution based on AspectJ extensions. AIRIA is an example of extension targeting the composition of aspects at a fine-grained level. It becomes possible to instrument the chain by inserting additional advices to store values, initialize flags, check conditions, etc. We demonstrated the feasibility of the instrumentation for various cases of interferences, exemplifying both data-and control-flow effects. In each case, we provided a working solution that was prototyped on randomly generated examples.

REFERENCES