Fine-Grained Implementation of Fault Tolerance Mechanisms with AOP: To What Extent?

Jimmy Lauret[1,2], Jean-Charles Fabre[1,2], Hélène Waeselynck[1,2]

[1] CNRS, LAAS, 7 Av. Du Colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, INP, LAAS, F-31400 Toulouse, France
{Jimmy.Lauret, Jean-Charles.Fabre, Helene.Waeselynck}@laas.fr

Abstract: The benefits of using aspect oriented programming (AOP) for separation of concerns is well-known and has been demonstrated in many works, including for dependable computing. In this paper, we use this composition capability of AOP to develop micro-aspects that can be combined together to realize a given fault tolerance mechanism. The toolbox of micro-aspects can be used to make mechanisms easily configurable and by the way to simplify their update. We show that the composition of micro aspects leads to undesirable side effects of the interactions between them, called interferences. We propose an approach to detect interferences with executable assertions, using an extension of AspectJ called AIRIA that enables control over an aspect chain at a shared join point. We finally draw the lessons learnt and discuss to what extent AOP can be used to develop fault tolerance mechanisms.

1 Introduction

The benefits of using aspect oriented programming (AOP) for separation of concerns is well-known and was demonstrated in many works, including for dependability. Mechanisms like *Time Redundancy* and *Control Flow Checking* have been implemented recently in AOP in the automotive context [1]. Fault tolerance mechanisms (FTM) can be implemented in separate software components, the aspects, and later woven at specific locations of the application code, called join points. In practice several mechanisms can be composed together to fulfill different fault tolerance properties. But composition may also be used to develop a single mechanism. For example, a replication mechanism to tolerate a server crash could be designed as the composition of several individual aspects, each of them realizing a small task: handling request sending to multiple destinations, handing request reception depending on the replica mode of operation, handling checkpointing, handling response return to client. This idea leads us to the notion of *micro-aspects*, which is similar to the notion of micro-protocols developed by R. Schlichting et al. [2-3]. It is appealing to resilient computing [4] since mechanisms become easier to configure and to update. By offering fine-grained, reusable micro-aspects, we intend to simplify the job of the developers. The first question we tackle in this paper is thus:

To what extent can fine grain aspects be defined and composed to implement reconfigurable fault tolerance mechanisms?

But the composition of the micro-aspects to realize a replication protocol may also lead to undesirable side effects, called interferences. In other words, the aspect chain might be incorrect after weaving all micro-aspects into the application code. We need to control the composition and provide means to detect undesirable interferences.

The second question we tackle relates to interactions between aspects and how undesirable interferences between aspects can be detected?

We propose to instrument the aspect chain using executable assertions. The proposed approach relies on the concept of *resolver* provided by an extension of AspectJ [5], called AIRIA [6].

The paper is organized as follows. In section 2, we present the interesting features provided by AIRIA to control the composition of aspects. In section 3, we develop a replication mechanism using a collection of micro-aspects. In section 4, we present our instrumentation approach to detect interferences. In section 5 we discuss the benefits and the drawback of using micro-aspects to develop fault tolerance mechanisms. Section 6 concludes the paper.

2 AspectJ and Variants

AspectJ [5] is our target language in this work. When several aspects are woven at the same join point [7], a given ordering is required. AspectJ offers a *declare precedence* statement for this. 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, assume an aspect *ACrypt* has two advices for encryption and decryption, and another aspect is used for logging messages. We would like to log clear messages, but it is not possible to declare separate precedence policies for logging/encryption on the one hand, and decryption/logging on the other hand.

Finer-grained resolution of conflicts can be found in AspectJ extensions, like the AIRIA extension [6]. It offers a *Resolver* construct to define precedence policies for advices. A resolver is a kind of around advice (cf. Figure 1) used to control the composition of advices at shared join points, i.e. an advice for composing advices. In Figure 1, the resolver sender is applied at each join point where the advices ALog.logMsg (message), ACrypt.encrypt (message ciphering), AAuth.auth (sender authentication), are woven (as specified in the and clause, line 3). The list between brackets determines the order of execution for the proceed clause (line 4). In the example, AAuth.auth is applied first, then Alog.logMsg is applied and finally ACrypt.encrypt encrypts the message before sending.

More complex policies can be defined by using if-then-else statements to select the appropriate proceed clause. Also, multi-level policies are possible with resolvers of resolvers. 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.

```
1  aspect LogEncryptAuth {
2  void resolver sender():
3  and(ALog.logMsg, ACrypt.encrypt, AAuth.auth) {          // pointcut selection
4  [AAuth.auth, ALog.logMsg, ACrypt.encrypt].proceed();}   // order before proceed
5  }
```

Fig. 1. The Resolver construct of AIRIA

We found resolvers 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. Based on the resolvers scheduling the conflicting advices of our micro-aspects, we can add instrumentation advices to the schedule and precisely control their placement in the execution chain.

3 Fine-Grained Development of FTM Using Micro-Aspects

The objective here is to determine a set of reusable micro-aspects to implement an exemplary replication protocol (PBR, *Primary Backup Replication*). In this simple example, we assume reliable point-to-point communication channels and that the system is synchronous. Some implementation details will be skipped for space limitation and also because they are of no interest to define micro-aspects. The example given can be seen as too simple, even naïve, but it is complex enough to illustrate interferences between micro-aspects during its development.

3.1 Client-Side Micro-Aspects

In our simple example, the client handles two communication channels with the primary (primary_adr) and with the backup (backup_adr) when the primary fails.

Any request includes a client_id and a request_number, forming a unique request id. An aspect named ANumbering that provides two advices, insert and remove, manages request numbers. Similarly, an aspect named AIdentifier that provides two advices, insert and remove, manages the client id.

A third aspect is responsible for the management of requests in progress, called ACacheManager, providing an advice addin: ACacheManager.addIn stores the request into a cache together with its corresponding timer value.

When the primary fails, the pending request (stored into the cache) must be resent to the *alive* replica (the former backup that is now the primary). The handling of failed requests is done by another aspect called ARequestResend. It provides an advice called reload: ARequestResend.reload retrieves the request into the cache and reloads it for retry. The pointcut expression for this aspect captures the exception raised when the timer expires. The aspect ATimer provides two advices start (time_window) and stop.

The various aspects defined above are mostly weaved at two shared join points, namely send() and receive(). Each aspect executes an elementary action. Their correct composition implements the expected client behavior (cf. 3.3).

3.2 Server-Side Micro-Aspects

A server replica has two operational modes, primary or backup. We define in this section the micro-aspects needed to implement them. The first action is to initialize the operational mode of the replica. An AspectJ field introduction is used to set up the mode for each replica. A micro-aspect named AServerMode is responsible for the initialization of the replica's mode. It offers a unique advice AServerMode.init.

Micro-Aspects in Primary Mode. The behavior in primary mode is as follows:

1) the client sends a request (request_number, client_id, request_body) to the primary server that receives it (unless the server replica crashes).
2) the primary server processes the request, captures the state of the computation at the end of the processing step, sends a checkpoint message (request_number, client_id, response_body, primary_state) to the backup that stores it, and, lastly, forwards the response to the client (request_number, client_id, response_body).

The algorithm presented in Figure 2 is simplified. The response to the client is part of the checkpoint message because it is necessary in case of primary crash. Suppose that the primary crashes after sending the checkpoint message and just before sending back the response to the client. When the timer corresponding to this request expires, the client re-sends the request to the new primary (the former backup). In this case, the request cannot be processed twice for conservative reasons (*Only Once Semantics*). To this aim, filtering duplicate requests is a mandatory.

When received, a copy of the request is forwarded to the backup that stores it into a cache. When the primary fails, the failure detector triggers the reconfiguration of the backup as a new primary. The new primary can process pending requests stored in the cache. The above described behavior leads to define the following micro-aspects to implement the primary behavior.

The management of the inter-replica protocol in this example is delegated to a micro-aspects named ACheckpoint that provides two advices:

- ACheckpoint.ForwardRequest forwards the request to the backup;
- ACheckpoint.BuildCheckpoint first captures the state of the replica and then prepares the checkpoint message (request_number, client_id, response_body, primary_state) before sending it.

```
1   variable of the protocole:        mode      // set to primary
2
3     receive request
4     if (mode=primary) {
5               forward request to backup
6               remove request number from the request
7               remove client_id
8               request processing            // new message only
9               checkpointing :capture state and send checkpoint to backup
10              insert request number in the response message
11              send response to client
12              }
```

Fig. 2. Pseudo algorithm of the primary

To remove the client_id and the request_id before processing the request (request_body), we use: AIdentifier.remove and ANumbering.remove.

Micro-Aspects in Backup Mode. The behavior in backup mode can be summarized as follows:

- the primary forwards a request (request_number, client_id, request_body) that is stored by the backup into a cache;
- the backup also receives the checkpoint messages from the primary (request_number, client_id, response_body, primary_state). The backup updates its current state with the primary_state information and stores the response into the cache (request_number, client_id, response_body).

The (request_number, client_id) is a unique index for sorting information into the cache that can be seen as a hash table.

The management of both the copy of the request message and the checkpoint message are done by the same aspect ACheckpoint previously defined. To handle both messages, we add three more advices in ACheckpoint:

- ACheckPoint.putInCache stores the request into the cache;
- ACheckPoint.updateCache stores the checkpoint into the cache;
- ACheckPoint.updateState updates the backup state with the primary state.

The receive statement in the backup mode is thus a join point for multiple advices depending on the type of message.

It is worth noting that, in the current implementation, the primary failure event raised by the failure detector triggers an event handler that changes the mode of the backup replica to primary (with no backup until a new backup is installed).

3.3 Composition of Micro-Aspects

Micro-Aspects Integration for the Client. The micro-aspects ANumbering.insert, ACacheManager.addIn, ATimer.start, AIdentifier.insert, are applied at the same shared join point, i.e. before the send statement. These are before advices to be applied in the following order: ANumbering.insert < ACacheManager.addIn < AIdentifier.insert < ATimer.start. The precedence order is determined by the resolver in Figure 3. The receive statement is also a shared join point. Figure 4 shows its resolver.

```
1  aspect sendResolution {
2  void resolver sender ():
3  and (AIdentifier.insert, ACacheManager.addIn, ATimer.start, ANumbering.insert) {
4  [ANumbering.insert, ACacheManager.addIn, AIdentifier.insert, ATimer.start].proceed (); }}
```

Fig. 3. Resolution of interactions around the send join point

```
1  aspect receiveResolution {
2  void resolver receive ():
3  and(ANumbering.remove , ATimer.stop){
4  [ANumbering.remove , ATimer.stop].proceed();}}
```

Fig. 4. Resolution of interactions around the receive join point.

Micro-Aspects Integration for Server Replicas. The behavior of the replica depends on the mode, primary or backup. We then use a *"resolver of resolver"* approach. The root resolver RCheckMode.getMode determines the mode and then the management of the appropriate advice chain is delegated to a child resolver. A resolver called RduplexPrimary.run manages the list of advices in primary mode. A resolver called RduplexSecondary.run manages the list of advices in backup mode.

The RduplexPrimary.run resolver applies the advices around the service in the following order: ACheckPoint.forwardRequest **<** AIdentifier.remove **<** ANumbering.remove **<** service **<** ACheckPoint.buildCheckPoint .

In backup mode, two series of actions must be made, first i) when the request is received, and ii) when the checkpoint is received. These actions are performed by the RduplexSecondary.run resolver:

- it determines first whether the received message is a request or a checkpoint;
- when it is a request the advice ACheckPoint.putInCache is invoked;
- when it is a checkpoint the advices ACheckPoint.updateCache and ACheckPoint.updateState are called.

Last but not least, the filtering of duplicate requests is delegated to the ACheckDuplicate micro-aspect that provides a unique advice named reply. ACheckDuplicate.reply first detects duplicate requests and, in this case, extracts the response, if any, from the cache. The response is returned to the client. When no response is retrieved in the cache, the request needs to be processed.

Table 1. Resolvers of resolvers summary

Level	Name	Triggered advice or resolver
Resolver level 0	RCheckMode.getmode	Resolvers level 1
Resolver level 1	RduplexPrimary.run RduplexSecondary.run	*Each resolver triggers the advices defined for each mode*

In summary, the final implementation uses two levels of resolvers, the behavior of each mode is handled by level 1 resolvers scheduling advices in each case. Level 0 is the root resolver handling level 1 resolvers depending on the mode of operation of the replica (cf. Table 1). Resolver level 0 (the root resolver) checks the mode and triggers RduplexPrimary.run or RduplexSecondary resolvers accordingly. Resolvers RduplexPrimary.run and RduplexSecondary trigger in turn the advices defined for each mode.

Discussion. In this section we have shown how micro-aspects can be defined and integrated together to realize a given replication protocol. This integration leads us to insert many micro-aspects at shared join points. The scheduling is managed by resolvers, even by resolvers of resolvers. The interesting point here is that many these micro-aspects are reusable for different variants of replication protocols (AIdentifier, Anumbering, ATimer, AReSend, ACacheManager), some being more specific (ACheckpoint). But, even in this simple replication protocol, subtle interactions can lead to interferences, i.e. errors. In Section 4, we propose an approach to prevent and detect interferences.

4 Interference Detection

An interference is an undesirable interaction between aspects woven at a shared join point. During the integration of several micro-aspects at a given join point, some assumptions regarding micro-aspects interactions are implicit, potentially leading to an interference. These assumptions must be made explicit. These properties are transformed into executable assertions in order to verify the composition at runtime.

4.1 The Interference Problem

Interferences are always defined by considering the sequential execution of aspects woven before, after or around a target instruction in the base code (the shared 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*). The authors of [8] consider four cases of interference, two dataflow and 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*. *A*'s behavior might differ from the one we 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 be inadequate, or partly cancelled.
- *Invalidation Before (IB)*. 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)*. 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*.

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 interaction 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*.

The important questions to be addressed are the following: How to detect interactions of multiples aspects at a shared join point? How to specify the expected behavior? How to validate composition?

4.2 Proposed Approach

The approach is described in detail in [9], we just summarize its main concepts and practical steps in this section. The aim is to detect interferences in the integration of micro-aspects into an application.

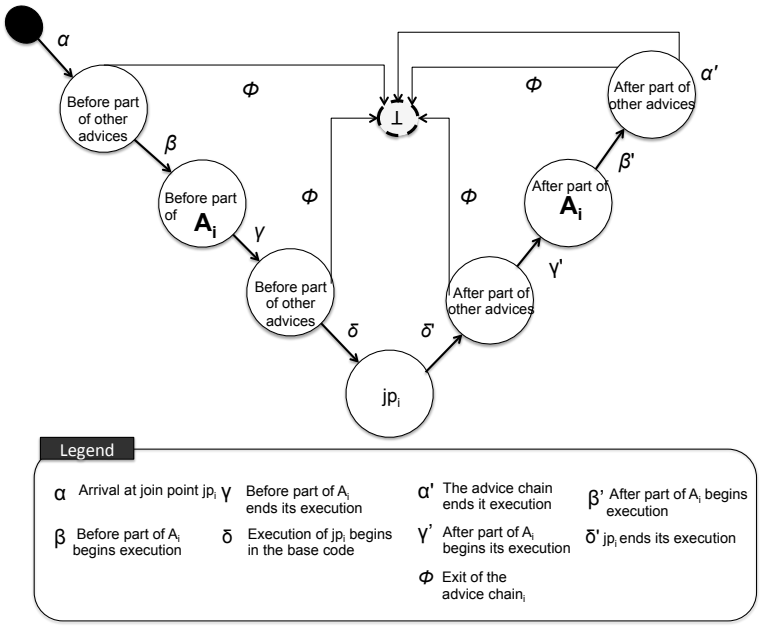

Fig. 5. Lifecycle of an around advice

Overall Principle. First, the designer must declare whether data- or control-flow interactions between a set of aspects are expected or not. Then, the resolution implies defining the correct order of the application of multiple advices at a shared join point. Interferences may induce subtle failures, we propose to reveal them by means of assertions. To keep separation of concerns, the assertions should be implemented with dedicated aspects that monitor the execution of other aspects. The *resolver* construct is used to this aim.

Figure 5 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 advices, 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 **α** represents the passing of control from the base code to the first advice, **β** 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 (**δ'**,..., **α'**). At any time, the control flow may get out of the chain of advices (**ϕ** 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 **α** and detect its change at **β**. Unlike AspectJ, AIRIA does expose transfers of control between the base level and the aspects, like **α** and **δ**, thanks to the *resolver* construct. In [9], we demonstrated that we can precisely control the placement of instrumentation code at any transition of Figure 5. We provided a working solution that was prototyped on artificial examples with randomly generated trees of resolvers. It allows us to automatically instrument the code to detect data- and control-flow interferences.

Detection of Data-Flow Interferences. The detection of both CB and CA consists of the use of two dedicated 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. For example, detection of a CB interference affecting the before part of a distinguished before advice $\mathbf{A_i}$ involves placing the storer at α and the checker at β. It is straightforward to apply a similar approach to the verification of properties attached to the after part of $\mathbf{A_i}$ (e.g., the values at transition γ' should be the same as at δ'), or even to properties spanning the before and after parts (e.g., the values at γ' should be the same as at α).

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 $\mathbf{A_i}$ is always executed after transition α. A first advice initializes the flag at α and another one sets the flag at β. Pieces of code placed in the root resolver expose a ϕ transition if the flag has not been set. Assume the execution chain is broken (e.g., an advice raises an exception, or does not call proceed to pass control to the next advice in the chain): whatever happens, control will come back to the root resolver so that we are able to detect the unset flag. IA detection follows a similar principle.

4.3 Interference Detection in the PBR Implementation

Properties Specification. We present first the expected properties for the correct integration of the previously defined micro-aspects to implement our duplex protocol variant. Our objective is to prevent CB, CA, IB, IA interferences among micro-aspects. We do not detail here the various steps of the interactive specification process but we show some of its result, i.e. the result of the analysis of micro-aspects to be combined at a shared join point.

Let's take an example, i.e. the client-side use of the advice ANumbering.insert at a send join point. The duplex protocol integrator must answer questions concerning the weaving of micro-aspects at a send join point.

For instance, a question can be: *"Is there any input variable whose value must not be modified from the join point to the execution of ANumbering.insert ?"*. The answer is *yes*: variable msgContent from the base code must not be changed before the execution of ANumbering.insert. The absence of CB interference must be checked. Such an analysis is applied to every micro-aspect used at a given shared join point.

Integration Properties of Client-Side Micro-Aspects. The properties to be addressed when combining micro-aspects used at the client-side are summarized in Table 2. Column 1 identifies the advice, column 2 the interference, column 2 the data, if any. Row 1 in the table corresponds to the ANumbering.insert advice example. A CA detection is attached to the AIdentifier.insert advice, Row 2. In our

implementation, it is important to insert first the request number, and then the client id. Row 3 and 4 relate to the advices that store the request in a cache and start a timer. If the request is stored and the timeout later occurs, but the request was actually not sent, we have an IA interference: we want to detect this incorrect behavior.

Table 2. Forbidden interferences for client-side micro-aspects

ANumbering.insert	CB	msgContent
AIdentifier.insert	CA	msgContent
ACacheManager.addin	IA	-
ATimer.start	IA	-

Integration Properties of Primary Mode Micro-Aspects. The properties to be addressed when combining micro-aspects used for the primary are summarized in Table 3.

For the advice AForwardRequest.forward, we must prevent a CB interference through msgContent: no modification of msgContent can be made before the execution of AForwardRequest.forward. In addition, we must verify that the join point is executed after the request has been forwarded to the backup. This is a possible IA interference for AForwardRequest.forward.

For the advice ACheckPoint.buildCheckPoint, we must prevent a CB (respectively CA) interference through msgContent: no modification of msgContent can be made before (respectively after) the execution of ACheckPoint.buildCheckPoint. Finally, we require that the client id is first removed from the message content, and the request number removed last.

Table 3. Forbidden interferences for primary-side micro-aspects

AForwardRequest.forward	CB,IA	msgContent
ACheckPoint.buildCheckPoint	CB, CA	serviceResult
ANumbering.remove	CA	msgContent
AIdentifier.remove	CB	msgContent

Integration Properties of Backup Mode Micro-Aspects. For the backup, the first interference is the same as for the primary. For all other advices, ACheckPoint.{putInCache, updateCache, updateState}, msgContent must not be modified before execution of the advice.

Thanks to this assertion checking approach, four integration faults have been successfully detected during the development of our replication mechanism, two genuine ones and two introduced on purpose.

5 Lessons Learnt

This idea of using micro-aspects like those defined in this paper can be interpreted as pure madness by the reader! In a certain sense it is! Many works in the last 20 years demonstrated that separation of concerns (with meta-objects, aspects, etc.) was an important paradigm for dependable computing. Since then, aspect-oriented programming moved this idea into practice. Separation of concern is an interesting concept, but non-functional actions in a program can correspond to tiny sets of statements. To what extend this idea is fine for resilient computing? Evolution of software including dependability mechanisms calls for a fine-grain development approach: small pieces can be updated or changed according to the needs. However, the composition of such small pieces may lead to undesirable side effects.

We have demonstrated in this work that aspect oriented programming can be used to develop resilient fault tolerance mechanisms. Using micro-aspects, one can easily understand that developing a variant of a duplex mechanism can be straightforward. An active variant of our passive replication can be done simply by just changing the ACheckPoint micro-aspect by a ASynchronize micro-aspect. In the active variant, both replicas are active and process input requests, only one replica replies to the client. The ASynchronize micro-aspect triggers the execution of the request at the backup as well, the primary sends a notification of request processing completion to the backup, and sends a response to the client (only in primary mode).

However, the use of micro-aspects may be error prone. In particular, it creates many potential sources of interferences between aspects.

Consequently, we have proposed an approach to prevent and detect undesirable interferences. AIRIA's resolver construct allows controlling the order of conflicting advices. It offers finer-grained control than *declare precedence* in AspectJ. Moreover, it forces a total ordering to be defined, hence preventing unspecified cases where the AspectJ compiler chooses an arbitrary order. Also, it offers all the observation points required to instrument a chain of advices. It makes it possible to automatically instrument the code with executable assertions, attaching non-interference requirements to the composition of advices. Undesirable interferences are detected 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-flow and control-flow effects. This approach was used to validate our micro-aspects based implementation of a duplex protocol.

The resolver construct of AIRIA was very beneficial to solve our problem, but in practice it was complex when using resolvers of resolvers. The instrumentation of an aspect chain is mandatory because the composition is error-prone. We are able to instrument automatically an aspect chain, including handled by resolvers of resolvers [9]. Nevertheless, simpler solutions should be investigated, as programming resolver of resolvers is not an easy task that should be error proof.

Micro-aspect based design and interference detection are the two sides of the same coin! The benefits depend on the capacity to validate the integrated mechanism, i.e. the composition of many micro-aspects. The interest of aspects for dependability is clearly depending on the validation capability we can propose regarding composition.

This applied to micro-aspects, at one extreme of the spectrum, but also to the composition of macro-mechanisms. Beyond interferences, point cut definition is a complex issue. For mechanisms like replication, the point cut is often simple (service calls) and should anyway remain simple to convince safety experts.

6 Conclusion

We have done the exercise of using AOP to develop a fault tolerance mechanism trying to take advantage or separation of concerns as much as possible. Using the micro-aspect approach was successful as it enables changing the protocol easily. However, flexibility is not free from a validation viewpoint. The detection of interferences between micro-aspects is a difficult problem. AspectJ does not provide a fine-grain control over aspect composition at a shared join point. This is why we used the resolver construct in the AIRIA extension. From our experience, a hierarchy of resolvers may be complex to use, but can be instrumented for detecting errors.

Our future work will be on the elicitation of conflict resolution policies and associated non-interference requirements. We envision a wizard tool aiding operators to enter scheduling directives and expected properties, before resolver code generation proceeds automatically. We will provide support for this, with an interfacing to the instrumentation solution already existing.

Acknowledgements. This work was partially supported by the IMAP project (Information Management for Avionics Platform) in collaboration with Airbus.

References

1. Alexandersson, R., Öhman, P.: Implementing Fault Tolerance Using Aspect Oriented Programming. In: Bondavalli, A., Vilar Brasileiro, F., Rajsbaum, S. (eds.) LADC 2007, LNCS, vol. 4746 , pp. 57–74. Springer, Heidelberg (2007)
2. Bhatti, N., Hiltunen, M., Schlichting, R., Chiu, W.: Coyote: a System for Constructing Fine-Grain Configurable Communication Services. ACM Trans. Computer Systems 16(4), 321- 366, ACM (1998)
3. Hiltunen, M., Taiani, F., Schlichting, R.: Reflections on Aspects and Configurable Protocols. In: Robert E. Filman (ed.) 5th International Conference on Aspect-Oriented Software Development, pp. 87-98. ACM Press (2006)
4. Laprie, J-C.: From Dependability to Resilience. In: Fast Abstracts of DSN 2008, Anchorage (2008) http://www.ece.cmu.edu/~koopman/dsn08/fastabs/dsn08fastabs_laprie.pdf
5. Colyer, M., Clement, A.: Aspect-Oriented Programming with AspectJ. IBM Systems Journal 44(2), 301–308 (2005)
6. Takeyama, F., Chiba, S.: An Advice for Advice Composition in AspectJ. In: Baudry, B., Wohlstadter, E. (eds.) SC 2010. LNCS, vol. 6144, pp. 122–137. Springer, Heidelberg (2010)
7. Hilsdale, E., Hugunin, J.: Advice Weaving in AspectJ. In: Murphy, G.C., Lieberherr, K.J. (eds.) 3rd International Conference on Aspect-Oriented Software Development, pp. 26-35. ACM Press (2004)
8. Katz, E., Katz, S.: User Queries for Specification Refinement Treating Shared Aspect Join Points. In: Fiadeiro, J.L., Gnesi, S., Maggiolo-Schettini, A. (eds.) 8th IEEE International Conference on Software Engineering and Formal Methods, pp. 73–82. IEEE Computer Society (2010)
9. Lauret, J., Waeselynck, H., Fabre, J-C.: Detection of Interferences in Aspect-Oriented Programs Using Executable Assertions. In: ISSRE Workshops 2012, pp. 165–170. IEEE (2012)