International Journal of Software Engineering and Knowledge Engineering

A Fine Grained Aspect Coordination Mechanism

Arturo Zambrano

LIFIA, Facultad de Informática, Universidad Nacional de La Plata, 50 y 115, CP 1900, La Plata, Buenos Aires, Argentina
arturo.zambrano@lifia.info.unlp.edu.ar

Silvia Gordillo*

LIFIA, Facultad de Informática, Universidad Nacional de La Plata, 50 y 115, CP 1900, La Plata, Buenos Aires, Argentina
silvia.gordillo@lifia.info.unlp.edu.ar

Johan Fabry†

PLEIAD, Computer Science Department (DCC), Universidad de Chile Blanco Encalada 2120, Santiago, Chile
jfabry@dcc.uchile.cl

Aspect interactions and conflicts are an important issue for AOSD. Aspectual conflicts are difficult to manage and, specifically semantic ones remain an open issue for AOSD. In this work, we present a simple mechanism for handling semantic conflicts at the granularity of aspect and advice in the resource awareness domain. We also analyse the impact of such coordination mechanism on some important properties of software (some of the so called *ilities*). We found that aspects can effectively be coordinated and remain independent, improving their reusability, modularity and evolvability.

1. Introduction

Aspect oriented programming (AOP) provides for a new and better modularisation mechanism for crosscutting concerns [12]. A much discussed and controversial feature in aspect oriented programs is obliviousness [11]. It started out as a key idea behind AOP, but now we realize that the relationships between *objects and aspects* and *aspects and aspects* are very complex, and total obliviousness is impossible –new and more restricted forms of obliviousness are used today. In any case, obliviousness is welcome as it promotes decoupling. Ideally, applications remain independent of the existence of aspects. At the same time, aspects should be as oblivious as possible of each other, keeping their independence and easing their reuse. The goal is to have independent aspects that are woven with the base application, resulting in the final (composed) desired functionality.

*also CICPBA
†Partially funded by FONDECYT project 1090083

Unfortunately, this is not always the case due to interactions of aspects. Such interactions may arise during the composition of aspects or at runtime. Generally, aspects are said to interact if they operate on the same joinpoints or if they structurally affect code of the same base module [6, 16, 19]. As pointed out by Monga et al. [14], these kind of interactions can be either be desired, e.g. if they are collaborations, or be conflicts. So, aspects can interact by collaborating (reinforcing) or conflicting [17]. However, aspects can interact even when they do not work on the same joinpoints. These are semantic interactions; their consequences can be observed through the side effects that aspects produce during its execution. In certain cases, as explained in section 2.2, an un-coordinated aspect activation can cause the system to misbehave. Therefore, an aspect might affect the behaviour of other aspects even when they are working on different joinpoints. As we will see, the occurrence of this kind of conflicts might depend on run-time conditions.

We have studied semantic aspect interactions in the field of resource awareness. Resource awareness [15] is a subdomain of context awareness [8]. Context awareness is crosscutting, since context related behaviour tends to be spread along several objects. It can be modularised using AOP, as is shown in [23]. The use of AOP for managing constrained resources is also advocated by Sousan et al. in [18]. In this domain, aspects can be used to control and optimise usage of scarce resources. However, in order to save some resource, an aspect may consume other resources which may be controlled by other aspects. In a previous work we have presented the bases for a coordination mechanism which allows to coordinate aspects activation [24]. In this work we present an extension that allows for a fine grained aspect coordination. It allows advice activations to be controlled in order to avoid or solve conflicts. Conflict resolution based on this mechanism has effects on software *ilities*. For example, solving semantic aspects interactions will increase reusability and adaptability, but may potentially negatively impact on comprehensibility. Other studied properties are: adaptability, evolvability, reusability, modularity, extensibility and scalability. We analyse each affected property to evaluate the mechanism.

This paper is organised as follows: section 2 presents both the joinpoint and semantic aspectual interaction problem. Section 3 motivates the need for a fine grained aspectual control and present the developed mechanism. Section 4 analyses the impact of the approach considering the software *ilities*. Section 5 reviews related work in the area. Finally, section 6 concludes the paper and presents some future work.

2. Aspect Interactions

In this section we will present joinpoint and semantic interactions, in order to give context to our work.

2.1. *Joinpoint Aspect Interactions*

Aspects can interact in several ways. The most frequently studied is by having intersections in their pointcut shadows. That is, if more than one pointcut shadow happens to affect the same joinpoint a potential interaction is present. This means that, at least two aspects, may affect the same joinpoint.

This kind of potential interactions can be detected at weaving time. Depending on the underlying AOP language or mechanism, it may be possible to expand the set of joinpoints affected by aspects, and check which jointpoints are being captured by more than one aspect. In that stage of composition, it is also possible to analyse how the execution context of the joinpoint is being accessed (observed or manipulated).

The weakness in this joinpoint approach is that it could lead to identify false conflicts. It is clear that, in many cases, having several aspects working on the same joinpoints is not harmful, and this scenario does not imply a conflict. For example, lets suppose there is a class whose instances are persistent and, for the same class, we define every invocation of a set method to be logged; we consider both concerns (Persistence and Logging) implemented as aspects. In this case, both aspects can overlap since changing some instance variable through a setter should be reflected in the DB and, as we said before, the setters need to be logged. The final composition of the base class and Persistent and Logging aspects will not produce conflicts even when they affect the same joinpoints, and possibly they access the same information (objects' internal state) from the execution context.

That category of interactions and conflicts, which here we call *joinpoint conflicts* has been the focus of research in the AOP community, as can be seen in [9, 6, 4]. There is however another category of interactions that are not detectable through joinpoint analysis, we call them *semantic interactions*.

2.2. *Semantic Aspect Interactions*

Semantic interaction and conflicts are not obvious, given that they can occur even when aspects do not work on the same jointpoints. They might arise as a consequence of a desirable feature such as low coupling: In order to keep coupling as low as possible, aspects should not make assumptions about the existence of other aspects. Thus aspects are more likely to be reused in isolation. Such independence between aspects will lead to code each one as if it were working alone on the base application. The behaviour of such aspects may affect objects in the base application, which are crosscut by other aspects, but not necessarily by working on the same joinpoints.

To illustrate this situation consider the following example, where conflicting aspects, implementing resource awareness, lead the system to an undesirable state.

2.2.1. *Memory Saver Vs Battery Power Optimiser*

Consider a system which runs in a mobile device and which is attached to a wireless network. In this context, suppose there are two aspects which deal with optimisations of use of memory and power.

Memory Saver Aspect monitors the memory usage by periodically checking the amount of free program memory. When it detects there is little memory available, this aspect forces all caches to flush their content.

Battery Power Optimiser Aspect is in charge of maximising the lifespan of the battery charge. Since wireless network connections consume a lot of power, this aspect delays such connections; that is, whenever the mobile client tries to send data to the server, the optimiser captures the outgoing data and stores it temporarily. When enough data has been collected, the optimiser creates a real network connection and sends all the stored data to the server.

Battery Power Optimiser affects the resources it needs to perform optimisations, mainly memory, which is also the focus of the *Memory Saver* aspect.

Both *Memory Saver* and *Battery Power Optimiser* are oblivious to each other, that is a desired property but, even though each aspect is aimed to work on its own concern, each one is influenced by the behaviour of the other (depending on dynamic conditions).

In a runtime scenario where there is few available memory **and** low battery power, aspects can enter in a vicious circle of activations (see figure 1):

(1) In order to free memory, the memory saver will try to empty buffers, which in the case of network buffers implies to do the real connections, thus using power. After spending power, the battery optimiser aspect could enter in action.
(2) The battery optimiser, in turn, will try to avoid doing connections, so it will attempt buffering data, thus spending memory. This may activate the memory optimiser aspect, there going back to 1.

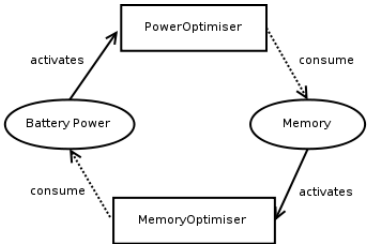

Figure 1. A cycle of resource consumptions and aspect activations.

Figure 1 illustrates this problem, ellipses are resources, and rectangles are aspects. Dotted arrows indicate consumption of resources. Solid arrows indicate a

resource is under the control of the aspect where the arrow arrives. An aspect in charge of optimising a resource is activated when its resource reaches some threshold level. In short, aspects are activated as side effects of the execution of other aspects.

Note that, in other scenarios where, for example, there is low battery power but enough memory for buffering, there is no conflict, because there is no problem in buffering as needed, and save some power. Or consider the opposite case, where there is little free memory but enough battery, in that case it is possible to transmit all the data in order to minimize the memory usage for buffering.

3. Aspect Control for Resource Awareness

3.1. *Prior Work*

In [24] we proposed an approach for semantic conflict resolution based on aspect coordination. This coordination is based on the idea of (de)activation of aspects according to the system's runtime context.

The target domain of this work was resource awareness. In order to optimise the use of a resource, another resource might eventually be affected, so it may imply that another aspect to get activated. To avoid aspects entering in a vicious circle of activations, it is necessary to coordinate their behaviour. However, having the aspects coded assuming the existence of other aspects is not feasible, since this would compromise its reusability.

In the mentioned work, we presented a simple mechanism for activating and deactivating aspects based on metadata describing aspects' properties. Such metadata is used along with information about the runtime context to decide if it is necessary to deactivate some aspect in order to avoid conflicts.

A downside of that work was that aspects can just be completely (de)activated. In certain situations, such a feature forces the developer to split a single concern into several aspects. Imagine an optimisation concern that can be implemented as an aspect with several advices. If just some of the advices produces an undesired side-effect, the full aspect containing it must be deactivated. This limitation can lead to split the implementation of a simple concern into several aspects (and the corresponding source code files), which might compromise its maintainability (an example is provided in section 3.3).

In the next subsections we present an approach, aimed at providing advice level coordination capabilities. This finer control allows parts of aspect behaviour (advices) to be active or not. As a consequence, a concern can be implemented as one aspect while its behaviour can be effectively coordinated at advice level.

3.2. *Running Example*

As a running example for this paper we use a number of aspects for resource awareness. In this field, aspects can be used to implement resource management policies

[7, 18, 23]. For each resource, there should be an aspect implementing resource management. In addition, resource management for each resource may involve several actions that need to be performed in order to optimise its consumption. These actions can be mapped to one or several advices.

The following list presents different policies that different resource optimiser aspects can apply:

3.2.1. *Memory Optimisation*

(1) Cache flushing: release memory used by flushing caches. This action may imply the consumption of other resources such as bandwidth in the case of network caches.
(2) Data compression: releases memory but consumes more processor cycles.

3.2.2. *Battery Power Optimisation*

(1) Caching: minimise the use of power intensive resource such as wireless network (releae bandwidth), but cosumes memory for the cache.
(2) Adaptive backlight power: decrease it to save battery power, no other resource is affected.
(3) Decreasing CPU frequency in order to save battery: it is a way of consuming processor cycles, since there will be less processing power available.

3.2.3. *Bandwidth Optimisation*

(1) Data compression: releases memory but consumes more processor cycles.
(2) Selecting the encryption protocol: SSL (for instance using DES) is secure but it is a verbose protocol (so network consuming), if bandwith is scarce a possible policy is to stop using encryption or replace it by a lightweight encryption algorithm. Simpler encryption also releases CPU cycles.

3.3. *Finer Aspectual Control*

The main limitation of our previous work is a coarse granularity in the aspect coordination system. As we stated before, it was only possible to (de)activate complete aspects. As we will show in this section, there are situations where parts of an aspect's behaviour conflict. In such situations, it may be unnecessary to deactivate the full aspect, since the advices that are not harmful can be executed.

Given the policies mentioned in the previous subsection, consider the following cases where conflicts arise just in parts of aspects' behaviour.

Table 1. Aspects effect on resources. R and C means that a policy respectively *releases* or *consumes* the resource named in the column. Conflicts are shown in bold.

		Resources			
		Battery	Memory	Processor	Bandwidth
Mem. Opt.	P 1	-	**R**	-	**C**
	P 2	-	R	C	-
Batt. Opt.	P 1	R	**C**	-	**R**
	P 2	R	-	-	-
	P 3	R	-	C	-
Band. Opt.	P 1	-	-	C	R
	P 2	-	-	R	R

3.3.1. *Case 1:*

A resource aware system running in a notebook. When the power wire is unplugged, the battery optimisation aspect is activated. In order to save power, it will activate the three policies stated above. However, if the user is performing some intensive processing task, it would not be desirable to decrease the CPU frequency.

In this case it would be useful to have *just part* of the power consumption policies active. On the other hand it is still desirable to have all the power consumption concern implemented as a single aspect, so that its modularity and cohesion are preserved.

3.3.2. *Case 2:*

If *Memory Optimiser* is running and *Battery Optimiser* comes into play after the power wire is disconnected, they may enter in conflict. Policy 1 of Memory Optimisation conflicts with policy 1 of Battery Optimisation (see the corresponding rows of table 1).

In table 1 we can see that several aspects and advices have different, potentially counterproductive, effects on resources. Aspects and the implemented optimisation policies are organised as rows, while resources are in columns. “R” and “C” means the policy releases or consumes the resource. Conflict between memory optimiser first policy and battery power optimiser is shown in bold.

Note that such policies are just part of the aspects implementing the memory and battery optimisation concerns. Other policies might not conflict at all. Therefore, it would be desirable to have conflicting policies switched off, but letting the non-conflicting ones as active policies.

Table 2. Summary of aspects and advices.

Aspect	Policy	Adv.#	Advice Details
Memory Opt.	Flushing	1	Flushes caches.
	Data Compression	2	One advice for compressing before storing and one for uncompressing data when needed.
Battery Opt.	Caching	1	One for caching before sending data through the network .
	Adaptive Backlight	1	Decreases backlight.
	CPU Freq.	1	Decreases CPU speed.
Bandwidth Opt.	Data Compression	2	One advise for compressing before sending data and one for uncompressing upon reception.
	Encryption Algorithm	1	Changes the encryption algorithm.

Table 2 shows the aspects, policies, number of advices per policy and a brief explanation about their behavior. As we mentioned before, policies should be (de) activated as needed. When a policy is implemented by more than one advice, the group of advices needs to be managed at once. In table 2 such policies have a number of advices greater than one (column Adv. #).

3.4. *Requirements*

For the sake of modularisation, it is good to model each optimisation concern as an aspect. Each of these aspects would have several advices, each (or groups) of them implementing some of the proposed policies. As we have seen, each policy affects resources in its own way.

We want to have an adaptive system where conflicting policies are deactivated (at least those needed for removing a conflict). So, the required granularity for conflict resolution is the policy which, at the implementation level, is mapped to an advice. Therefore, we conclude that aspectual behaviour coordination should be performed at advice level.

Note that resource awareness policies may be split along several advices and methods. One advice may cope with some part of the optimisation policy, and others can complete the work started by the first one. In our previous example, the battery optimiser captures some data before it is actually sent through the network, and when the buffer is full, it sends all data; this behaviour can be clearly divided in two parts, one is the advice which captures data and the other, a method or advice where data is actually dispatched to the network. Therefore we need to manage groups of related advices and methods, this is another important requirement. To summarise,

```
1  @AffectsMemory(CONSUMED)
2  @AffectsBattery(RELEASED)
3   public aspect BatteryOptimiser
```

Listing 1. Excerpt of an annotated Aspect

the requirements for a conflict resolution mechanism in resource awareness are the following ones:

- To be capable of handling semantic conflicts based on runtime context.
- To solve conflicts at aspect granularity.
- To solve conflicts at advice granularity.
- To solve conflicts for groups of advices which implement a policy (group granularity).

Additionally, from the software *ilities* perspective we desire to keep aspects independent in order to facilitate its reuse, to allow the programmer to express the semantic of new aspects (comprehensibility and extensibility), and to keep the coordination behaviour modularised for evolvability.

3.5. *Design*

Figure 2 shows the components involved in the the proposed solution. Aspects (1) are enriched with metadata (2) indicating how they affect the resources (3) – which are shared by the base system and the aspects. Such metadata is added through annotations; we call them *semantic labels*. Each semantic label denotes a resource and an effect that the tagged aspect has on the resource. Having such semantic information allows us to write coordination logic for aspects. Coordination logic is expressed in rules, that indicate which aspect(s) must be (de)activated according to the current state of resources. But, instead of referring them by their names, rules use semantic labels. We show how they are used in the implementation section. The combined use of labels and rules for controlling aspects, avoid the coupling between aspects themselves and between rules and aspects. There is a coordination module (4) which takes care of monitoring the resource states (5) and evaluating the rules as necessary.

3.6. *Implementation Considerations*

In order to provide advice level control we extended our previous work so that annotations containing semantic information can be applied to advices and group of advices. When an aspect is instantiated, it is scanned looking for semantic labels. The semantic information found in the aspects is used to feed our coordination module, as depicted in figure 2. Code listing 1 shows how an aspect code looks like when semantic labels are present.

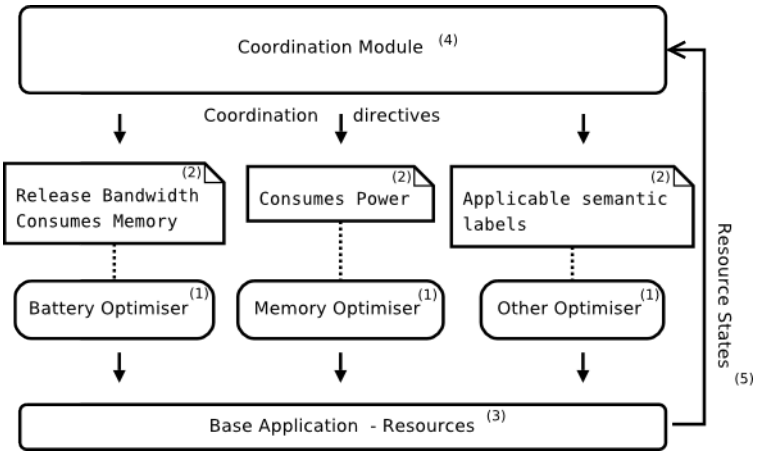

Figure 2. Aspects with their metadata (semantic labels) as comments, and coordination module.

```
1    when
2      m : Resource(name == "memory",
3                          availability < MEM_LIMIT)
4      b : Resource(name == "battery",
5                         availability < BATT_LIMIT )
6    then
7      Coordinator.stopConsumersOf("memory");
```

Listing 2. A simple rule for controlling aspects. This rule solves the conflict between Memory Optimiser Policy 1 and Battery Optimiser Policy 1

The inspection of semantic labels is carried out using the annotation reflection mechanism provided by Java. The scope of a semantic label is limited to the element where it is applied (that is aspect, advice or advice group). Aspectual semantic information is organised during loading time, so aspects and individual advices can then be easily referenced. In order to provide flexibility for expressing the rules, we decided to use a *rule engine*, we choose JBoss Drools [1] as it is a mature engine which allows for the definition of new rule languages – this feature can be used to develop rule languages targeted at specific domains..

Conditions, in our rules, contain expressions which refer to the execution context (for instance the availability of some resource). Actions express operations to be carried out on aspects, advices or group of them. For example, the code listing 2 shows a very simple rule written using JBoss Drools syntax. Lines between keywords `when` and `then` indicate a condition that should be satisfied in order to execute de code after the keyword `then`. Thes rule in Listing solves the conflict between Memory Optimiser(3.2.1) Policy 1 and Battery Optimiser (3.2.2) Policy 1, in this case favouring power management over memory usage .

The execution context is monitored, when a resource state changes, these

Table 3. Annotations, meanings and parameters for the running example. Note that new annotations can be defined to indicate affects on new resources.

Annotation	**Meaning**	**Parameters**
`@Group`	denote the advice as part of a group	the name of the group
`@AffectMemory`	indicates the advice or aspect affects memory availability	`Release` or `Consume`
`@AffectBandwidth`	indicates the advice or aspect affects bandwidth	`Release` or `Consume`
`@AffectPower`	indicates the advice or aspect affects battery power	`Release` or `Consume`
`@AffectProcessor`	indicates the advice or aspect affects CPU cycles	`Release` or `Consume`

changes are asserted into the rule engine, and coordination rules are (re)evaluated as necessary. According to the execution context, aspects and advices are activated or deactivated by the rules. Conflicts are avoided, since aspects or advices are active only if the execution context provides the right conditions.

In order to satisfy the requirements stated in 3.4, semantic labels can be applied to aspects, advices or groups of advices. So, when a rule express some action on a given role (for example *switch off memory consumers*) it affects the behavioural elements mentioned before. Note that the rule presented above does not (explicitly) reference any aspect or advice. It just refers to the role played by behavioural elements (for instance *memory consumer*).

3.6.1. *Groups of Advices*

For this work we introduce advice and group level coordination based on semantic labels. As semantic labels are annotations, aspect and advice scope is provided by the language support. In order to define groups of advices for coordination purposes, we provide a new annotation. The `@Group` annotation allows the developer to denote several pieces of behaviour as belonging to the same conceptual unit (an optimisation policy in our case). For example, an advice can be enriched with the group id which it belongs to: `@Group("NetworkBuffering")`. Semantic labels found in any member of the group are assumed to be applicable to all the group. Our `@Group` annotation is a workaround that manifests the need for other aspect composition facilities. Our intention is to provide the flexibility for expressing the policies and coordinate them.

Advice and group level annotations can override aspect level annotations. In contrast, advice annotations cannot override group level ones. Single annotated advices and groups are equivalents, since they are the implementation of a policy

```
1  @AffectsMemory(ResourceUsageType.RELEASED)
2  public aspect MemoryOptimiser extends ResourceOptimiser {
3       // pointcut definitions
4      @Group(''DataCompression'')
5      @AffectsProcessor(ResourceUsageType.CONSUMED)
6      void around(BaseSystem system): usage(system) {
7        //compress data code();
8    }
```

Listing 3. Aspect with advice annotations

– a policy may be implemented as one or many advices. So, trying to override group level annotation is similar to having two contradictory annotations on the same advice (which makes no sense). For example, in the previous explained example of a networking optimisation formed by two advices – one caching data and the other sending it when it is necessary – , the whole policy must be tagged as `@AffectMemory(CONSUME)`, it makes no sense to override this for the advice which sends the data (as `@AffectMemory(RELEASE)`), since whole policy is a memory consumer.

As labels are applied to any advice of the group and as valid considered for whole group, it is necessary to choose where to apply them, it seems reasonable to choose the more significant piece of behaviour (the most important advice in the group), in this case that which actually caches data.

So, if most advices of an aspect need the same annotation it can be defined at aspect level and redefined for the advice which does not follow the pattern.

Table 3 shows the annotations used for denoting aspects effects (just for the examples mentioned in this work) on resources (semantic labels) and the annotation for grouping. New semantic labels can be added to express effects on new resources which need to be taken into consideration.

Code listing 3 shows an aspect where annotations have been applied at aspect and grouplevel. Note that the group level annotation (line 5) complements the information asserted by the aspect annotation.

3.7. *Tool Considerations*

The current prototype is implemented in Java and AspectJ [3], and relies on JBoss Drools [1] as the rule engine. The use of these widespread tools, provide us with the possibility of testing it in a variety of environments. This is part of our future work.

The current version of our prototype depends heavily on the ability of the language to attach metadata to objects (aspects in our case). This approach could also be used in languages where such support is not available, an alternative implementation could rely on expressing metadata for aspects in a separate file (let say an XML file which describes the metadata for aspects and advices). Such alternative implementation would negatively affect the comprehensibility of the aspectual view

of the system, since aspects and its corresponding metadata would be physically separated. Another feasible approach would be including the semantic information in the comments, as it can be done using XDoclets [2].

The tools used for our proof of concept do not allow us to deploy aspects dynamically. Such a feature would certainly improve the adaptability (see section 4.1), letting us weave new aspects or unweave those which are known to be unnecessary.

4. Implications of our Approach

In this section we briefly review how this approach impacts on several properties of software. Note that the presented approach works for non-core functionality aspects, for sure that de-activating some core functionality would break requirements of our system.

4.1. *Adaptability*

The main objective of the aspect coordination approach we present here is to get context-based adaptability for aspects. As the system state evolves, it is monitored and aspects are activated according to the current context. Aspect behaviour is not actually adapted, instead, those aspects that are most appropriate for a given context are activated, and those which are potential source of conflict (in such a context) are deactivated.

4.2. *Evolvability*

Evolvability is the ability of a software artifact to cope with new requirements (or changes in the existing ones). Having the aspects separated from the conditions that must be met for running them, helps evolve these two concerns in an orthogonal way. Assuming there is a system which is working using the presented approach, if a new optimiser must be added, instead of checking all the needed runtime conditions in the aspect's code, it is sufficient to tag it properly and the rules will do the rest. On the other hand, if there is a new coordination requirement like *when the user is doing heavy processor tasks, bandwidth cannot be optimised using data compression, since it consumes a lot of processor cycles*, this new constraint can be added adding a new coordination rule, without altering any optimiser aspect.

The presented approach allows for an improved evolution of the whole system, specially for aspects. Aspects containing the logic of the crosscutting concerns are separated from the *conditions* which indicates if they should or should not be applied. Encapsulating such conditions in rules allows us to coordinate the behaviour of the whole system without touching aspects' source files. Adding new rules or modifying such conditions does not involve recompiling the aspects and weaving the system.

Another interesting effect of this approach is that rules can handle new aspects without the need of being recompiled, because they refer to the role of aspect. When

```
1  aspect MemoryOptimiser
2     pointcut usage: (@where)  &&
3              if(Battery.remaining()>LOWER_BATTERY_LIMIT)
```

Listing 4. Memory Optimiser aspect code, coupled variant 1.

```
1  aspect MemoryOptimiser
2     pointcut usage: (@where)  &&
3                       if(BatteryOptimiser.isInLowestLimit())
```

Listing 5. Memory Optimiser aspect code, coupled variant 2.

```
1  /*import  ...*/
2  @AffectsMemory(ResourceUsageType.RELEASED)
3  @AffectsBandwidth(ResourceUsageType.CONSUMED)
4  public aspect MemoryOptimiser extends ResourceOptimiser{
5     pointcut usage(BaseSystem system):  target(system)
6           && call(public void BaseSystem.systemUse(..));
7     public MemoryOptimiser()      { /* ... */}
8     void around(BaseSystem system): usage( system){
9           /*makes system buffers flush*/
10    }
11 }
```

Listing 6. Memory Optimiser aspect code, final version

a new aspect is added, it needs to be conveniently tagged, and that is enough for the coordination mechanism to handle it.

4.3. *Reusability*

Consider the example shown in section 2.2, where the MemoryOptimiser aspect cannot run if there is low battery power remaining (in this case BatteryOptimiser is more important). Writing aspects that deal with such situations by themselves compromise their reusability. For example, the MemoryOptimiser could look like code listing 4. Note that in this case MemoryOptimiser is being coupled to the battery power management logic, this is because it cannot run under certain power conditions. Another possibility would be to ask the BatteryOptimiser aspect as shown in the code listing 5. In any case the reusability of the MemoryOptimiser is affected since parts of their code refer to the battery management code. Specifically, in the second case the MemoryOptimiser aspect cannot be compiled without the BatteryOptimiser.

In the approach we present the MemoryOptimiser aspect does not refer to the BatteryOptimiser (or other aspects), neither check if it is possible to run according to the remaining battery power. As it is shown in listing 6 our MemoryOptimiser code is clean of references to other aspects, it just indicates how resources are used

by it. Every check regarding resource states is contained in the coordination rules, which are separated from aspect's code. In this way aspect reusability is improved, since coordination behaviour is completely decoupled from them. Therefore, every aspect is independent from each other, allowing aspects to be reused independently. Besides this, rules might be reused independently of aspects, since they have no direct references to aspects. A rule can handle any aspect correctly enriched with the metadata it needs.

4.4. *Comprehensibility*

Comprehensibility can be arguably compromised. On the one hand, it can be said that having aspects separated from code conditioning its execution affects negatively the comprehensibility of them, that is, reading an aspect source file it is not possible to know when its advices will actually be executed. On the other hand, it can be said that aspects remain correctly modularised and the *inter aspect coordination concern* has been factored out, so coordination rules are expressed separately. If the developer needs to read the implementation of a crosscutting concern, he/she should read the aspect's source file. If, besides this, she/he needs to understand the coordination schema, it is necessary to read the rules, which are not spread along aspects.

Comprehensibility of the aspects is improved as semantic information is also expressed in the aspects. This information allows the source code reader to understand the effects of the aspect in the base system.

4.5. *Modularity of Coordination Concern*

As we discussed in section 4.3, if each aspect implements the necessary checks to ensure that runtime conditions are the expected for its execution, the resulting code would be tangled with the coordination behaviour. This is because the coordination concern is inherently crosscutting: the necessary conditions for running a given resource optimiser, are a combination of the resource states.

Our coordination module (see fig. 2) is in charge of monitoring resources and controlling the execution of aspects (or advices/groups), in this way aspect does not contain coupling code as shown in listings 4 and 5 It can be argued that metadata indicating aspects roles is, in a way, part of a crosscutting concern spread along aspects. On the other hand it is possible to inject the annotations using intertype declarations in this way the optimisers code will be clean of labels.

4.6. *Extensibility*

New semantic information can be added by defining new annotations (following certain conventions), then this information is available for the definition of new coordination rules. As we said before, new aspects are automatically coordinated provided they have been conveniently annotated.

4.7. *Scalability*

Scalability has yet to be studied. There is no explicit limit for the amount of aspects or semantic labels that can be handled. In our opinion, the key factor for tuning performance is the number of rules and how often they are evaluated.

5. Related Work

In this section we review some related work as an attempt to determine the position of this work among others in the area of aspect interactions.

In [17], Sanen et al. present a classification of aspect interactions, which includes: conflicts (semantical interference), dependencies (aspects that need other aspects), reinforcement (aspects influencing correct working of others) and mutex (interaction type of mutual exclusiveness). Our work is oriented to the conflicts category, and writing the correct rules it could be applied to get mutual exclusion for aspects.

Douence et al. [10] propose a theoretical analysis framework to detect conflicts. In that work, aspects interactions can be detected through static analysis of the joinpoints. When conflicts are detected they can be solved by specifying the desired composition. Our approach does not depend on the fact of having aspects working on the same joinpoints.

Tessier et al. present, in [21], a model-based methodology which allows the detection of direct conflicts between aspects. In that work a taxonomy of conflicts is presented; the categorisation includes *Crosscutting Specifications*, *Aspect-Aspect Conflicts*, *Base-Aspect Conflicts* and *Concern - Concern Conflict*. Our work can be framed by the last of these categories, in particular by the subcategory *Inconsistent Behaviour*, that refers to conflicts where one aspect can alter the state used by other aspects. While Tessier's work is oriented to detection and resolution of conflicts by the developer in early stages of software life cycle, ours is aimed at solving conflicts at runtime.

Whittle et al. [22] discuss how to cope with some aspectual conflicts in model-driven approach. This work describes heuristics to drive the composition and solve conflicts. Such directives can define precedence between aspects; rename, add or delete model elements. Despite the fact that this approach is intended to work at the model level, and ours is for runtime, it is possible to get similar effect (adding and removing behaviour) based on the execution context.

Bergmans [5] proposes the use of annotations as a means of detecting conflicts among cross-cutting concerns. In his approach, conflicts can be detected when multiple concerns works on the same joinpoint. As we previously said, our work aims to solve conflicts arising even when the aspects involved work on different joinpoints.

Tanter et al. [20] propose the extension of aspect languages in order to support the notion of *context*. They present different examples of context, and show how aspects can make use of contextual information for their activation. Both works are similar regarding the use of runtime context, but they differ in that the approach presented by Tanter is more expressive, since contexts are first class entities,

allowing them to refer to context in pointcut expressions. In contrast, our "context dependent" expressions (rules) reside separately, which allow them to evolve independently from aspects.

As discussed in [13] by Kiczales et al. annotations are very useful for describing what is *true* about jointpoints (these kind of annotations are called there annotation-property). In our case, we use annotations to denote the effect of aspects on the system. Since they represent semantic (and design) information we decided to keep them tied to aspects.

As far as we are concerned, no work has been performed on the resolution of semantic conflicts at runtime.

6. Conclusions and Future Work

Conflict resolution is still an open issue for the AOSD community. Mechanisms for handling interactions between aspects provided in languages, such as AspectJ, leave much room for improvement.

In this paper, we have presented an extension of a mechanism for coordinating aspects' behaviour in order to avoid or solve conflicts, based on semantic information. This mechanism enables a fine grained control of the aspectual behavioural elements. We have discussed the impact of the approach under the light of some interesting software properties. It has been explained how this approach provides a flexible coordination behaviour while maintaining aspects independence. From the point of view reusability and evolvability, we have shown that this approach helps to use and evolve aspects and coordination behaviour in a separated way. What is more, having aspects enriched with semantic metadata improves its comprehensibility. From this analysis we conclude that the presented approach delivers more advantages than drawbacks and that it deserves further study. Part of our future work involves studying if this kind of mechanisms can be embedded in domain specific aspect languages, and analysing the possible benefits of such integration. Having an aspect language tailored for a given domain, would allow to reason about aspect interactions, improving interaction detection and resolution capabilities. Furthermore, domain specific conflict resolution strategies can be built into the aspect language.

Bibliography

[1] Jboss rules engine. http://www.jboss.com/products/rules.
[2] Xdoclet: Attribute-oriented programming. http://xdoclet.sourceforge.net.
[3] AspectJ project. http://www.eclipse.org/aspectj/.
[4] S. Bakre and T. Elrad. Scenario based resolution of aspect interactions with aspect interaction charts. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 1–6, New York, NY, USA, 2007. ACM.
[5] L. M. J. Bergmans. Towards detection of semantic conflicts between crosscutting

concerns. In J. Hannemann, R. Chitchyan, and A. Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

[6] S. Casas, C. Marcos, V. Vanoli, H. Reinaga, L. Sierpe, J. Pryor, and C. Saldivia. Administración de conflictos entre aspectos en aspectj. In *Proceedings of the Fourth Argentine Symposium on Artificial Inteligence*, pages 1–11, 2005.

[7] A. Dantas and P. Borba. Developing adaptive J2ME applications using AspectJ. In *Proceedins of VII Brazilian Symposium on Programming Languages, SBLP 2003*, pages 226–242, May 2003.

[8] A. Dey. Understanding and using context. In *Personal and Ubiquitous Computing Journal*, volume 5, pages 4–7, 2001.

[9] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *1st Conf. Generative Programming and Component Engineering*, volume 2487 of *lncs*, pages 173–188, Berlin, 2002. Springer-Verlag.

[10] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, Mar. 2004.

[11] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[13] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[14] M. Monga, F. Beltagui, and L. Blair. Investigating feature interactions by exploiting aspect oriented programming. Technical report, Dip . Elettronica e Informazione; Politecnico di Milano, 2003.

[15] D. Preuveneers and Y. Berbers. Towards context-aware and resource-driven self-adaptation for mobile handheld applications. In Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo, editors, *SAC*, pages 1165–1170. ACM, 2007.

[16] J. Pryor and C. Marcos. Solving conflicts in aspect-oriented applications. In *Fourth Argentine Symposium on Software Engineering*, 2003.

[17] F. Sanen, E. Truyen, B. D. Win, W. Joosen, N. Loughran, G. Coulson, A. Rashid, A. Nedos, A. Jackson, and S. Clarke. Study on interaction issues. Technical Report AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, Katholieke Universiteit Leuven, 28 February 2006 2006.

[18] W. Sousan, V. Winter, M. Zand, and H. Siy. Ertsal: a prototype of a domain-specific aspect language for analysis of embedded real-time systems. In *DSAL '07: Proceedings of the 2nd workshop on Domain specific aspect languages*, page 1, New York, NY, USA, 2007. ACM.

[19] É. Tanter. Aspects of composition in the reflex aop kernel. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2006.

[20] É. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006), at ETAPS 2006*, Mar. 2006.

[21] F. Tessier, M. Badri, and L. Badri. A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In M. Huang, H. Mei, and J. Zhao,

editors, *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, Sept. 2004.

[22] J. Whittle, J. Araújo, and A. Moreira. Composing aspect models with graph transformations. In *EA '06: Proceedings of the 2006 international workshop on Early aspects at ICSE*, pages 59–65, New York, NY, USA, 2006. ACM Press.

[23] A. Zambrano, S. E. Gordillo, and I. Jaureguiberry. Aspect-based adaptation for ubiquitous software. In F. Crestani, M. D. Dunlop, and S. Mizzaro, editors, *Mobile HCI Workshop on Mobile and Ubiquitous Information Access*, volume 2954 of *Lecture Notes in Computer Science*, pages 215–226. Springer, 2003.

[24] A. Zambrano, T. Vera, and S. Gordillo. Solving aspectual semantic conflicts in resource aware systems. In W. Cazzola, S. Chiba, Y. Coady, and G. Saake, editors, *Third ECOOP Workshop on Reflection, AOP and Metadata for Software Evolution*, Nantes, France, 2006.