Solving Aspectual Semantic Conflicts in Resource-Aware Systems

Arturo Zambrano
LIFIA
Facultad de Informática
UNLP
Argentina
arturo@sol.info.unlp.edu.ar

Tomás Vera
LIFIA
Facultad de Informática
UNLP
Argentina
tomasv@sol.info.unlp.edu.ar

Silvia Gordillo
LIFIA
Facultad de Informática
UNLP
Argentina
gordillo@sol.info.unlp.edu.ar

Abstract

Aspects sometimes conflict between them in scenarios where they reify resource awareness concerns. These conflicts are the result of the scarcity of resources and the fact that, frequently, aspects are mutually oblivious. This kind of conflict can be solved by managing aspects according to the context. Obliviousness can be retained and, at the same time, specific (per situation) conflict resolution strategies can be applied.

Keywords

Aspectual conflicts, resource-awareness, metadata, aspect management

1 Introduction

Mobile applications must face a continuously changing environment. Resource and service availability can change dramatically during run-time. Then, resource-awareness is a must in such applications. Context and resource awareness are *per se* invasive, and prone to produce tangled designs and code. Because of this, a common practice is to isolate such behaviour in the middleware layer. In [7] and [16] it has been suggested that the use of aspect orientation constitutes a means of decoupling context aware functionality from mobile applications.

The advanced separation of concerns provided by AOSD fits in the field of middleware in general [1, 2, 3], and it is specially suitable for implementing middleware for mobile context-aware systems. Aspects reifying different resource related concerns provide a modular way of handling them. On the other hand, aspects in resource-awareness middleware often compete for common (shared) resources they need. Conflicts are usual in this domain [14] and lead to aspect interactions which are related to the aspects' behaviour semantics (the way resources are utilised by aspects). These interactions cannot be detected just by syntactic means. As a result, they cannot be detected in compile time, since they depend on run-time conditions. Consequently, new approaches are needed to cope with this kind of conflicts.

In this work we will exemplify conflicts between aspects in a resource aware environment. We also present our approach for semantic-conflict resolution. This is based on the use of meta information attached to aspects, such metadata is used afterwards for conflict detection and aspect management.

This paper is organised as follows: Section 2 presents the motivation for our work. Section 3 summarises previous related research works. In Section 4 a conflict resolution mechanism is proposed. Finally, we present our conclusions and future work in Section 5.

2 Aspectual Semantic Conflicts

2.1 Context

In an aspectual middleware for mobile applications, several aspects adapt application's behaviour to run-time conditions to ensure they use the available resources in the most effective way. This approach releases the application of resource management responsibilities, modularising behaviour that otherwise would be tangled with application's one.

We argue that even when no interference is a desirable state for aspects, this condition is not always held in runtime; since aspects for mobile client-side middleware implement a concern consuming (possibly) shared resources. Examples are provided in section 2.2.

It can be said that aspects should be represented in such a modular way that they do not affect other aspects or concerns. On the other hand, any behaviour added by aspects will consume resources; at least it will consume the processing cycles needed to execute its instructions. In some cases, this is not a problem, but in the context of mobile computing, where resources are scarce, a resource-conflict[1] could be a major problem. Therefore, it is a pragmatic problem that has to be considered when several aspects that manage resources are running in a mobile middleware.

2.2 Examples

In this section we will exemplify some conflicting scenarios in the context of mobile computing. The examples show several aspects which interfere with each other while trying to accomplish their objectives.

2.2.1 Memory Saver Vs Battery Optimiser

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

[1] In the context of this work, a conflict means the use of a given resource in an uncoordinated way, which may be dangerous for a system

Battery Optimiser Aspect is in charge of maximising battery lifespan. 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 performs a real network connection and sends all the stored data to the server.

As the reader can see, *Battery Optimiser* is affecting the resources it needs to perform optimisations, mainly memory, which is also the focus of the *Memory Saver* aspect.

Both *Memory Saver* and *Battery Optimiser* are oblivious to each other. Even though each aspect is aimed to work on its own concern, each one is influenced by the behaviour of the other.

It is important to note that the kind of conflict we are dealing with cannot be determined in compile or static weaving time.

2.2.2 Network Optimiser Vs Security Vs Processing Time

Suppose that in the context of a wireless network, low traffic is desired in order to minimise packet loss problems and improve the response time. Also, as it is a wireless network, some security mechanism is needed, namely encryption. Conversely, encrypted messages are usually larger than their non-encrypted counterparts; therefore, they increase network traffic. Consequently we can have security but paying with a higher bandwith usage. In order to lighten this problem, we decide to compress messages before sending and decompress after receiving them. Now we have paid with CPU time.

A more sensible approach could be an adaptive one, where security is always provided using encryption, and compression is applied just when there is excessive network traffic and idle processor time.

2.2.3 Discussion

From the previous paragraphs it is clear that conflicts among aspects exist, even if they are not working on the same joinpoints.

While considering aspect conflicts, it is important to keep in mind the notion of obliviousness [9]. Despite the fact that obliviousness is not a requirement for an aspect oriented system, it is an important property that, if it is reached, it brings additional loose coupling. In this work we intentionally try to build context aware aspects that are mutually oblivious.

We argue that semantic conflicts can be solved by expressing the aspects' semantic without losing obliviousness. Aspects' semantics can be denoted through metadata; and modern programming languages, such as Java and C# provides means of expressing it. Therefore, it is fairly possible to develop metadata-based approaches, which can be easily implemented using the mentioned facilities.

3 Related Work

Recent research work in aspectual conflict detection has been developed by matching pointcuts syntactically[5]. Dounce et al. [8] propose a theoretical analysis framework to detect conflicts. However, little has been said regarding how to solve or avoid semantical aspectual conflicts; i.e. conflicts arising from the composition of behaviours that do not fit or are counterproductive, even when they might act on different joinpoints.

Concern Oriented Requirement Engineering techniques face the problem of conflicting concerns [12] at requirements level. In that work requirements are grouped into concerns, and the impact derived from the relationship between concerns is calculated. This impact is used to determine the existence of conflicts which can be solved by priorisation or renegotiation with the stakeholders. This approach may lead to a coarse grained aspect priorisation. In addition, this priorisation is fixed and applied to the whole system; since no specific situation customised priorisation is given. As it has been shown in Section 2.2, some conflicts may arise on runtime, and the time of their occurrence cannot be foreseen during requirements engineering phase.

Tessier et al. present, in [15], a model-based methodology which allows the detection of direct conflicts between aspects. In that work a taxonomy of conflicts is offered; the categorisation includes *Crosscutting Specifications*, *Aspect-Aspect Conflicts*, *Base-Aspect Conflicts* and *Concern-Concern Conflict*. Our work can be framed by the later category, in particular by the subcategory *Inconsistent Behaviour*, that refers to conflicts where one aspect can alter the state used by other aspects.

Bergmans [4] propose 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 join point. As we previously said, our work aims to solve conflicts arising even when involved aspects work on different joinpoints.

This work is different with respect to previous conflict resolution approaches because it is focused on the semantic and behaviour of aspects, rather than their syntactic pointcut clashing.

4 Semantic Conflict Resolution for Resource-Awareness

4.1 Analysing Conflicting Situations

Despite the fact that conflicts cannot be completely foreseen using syntactic techniques, it is possible to anticipate conflicting situations by performing a domain analysis and reasoning about risky system situations.

In the context of *resource-awareness*, conflicting situations can be characterised as "malformed" combinations of resources' states. By "malformed", we mean situations where aspects executing normally, but in a non-coordinated way, can affect the proper system's behaviour. Following our first example, a conflicting situation arises when the system has little memory, and data cached by *Battery Optimiser* must be flushed very quickly. In this case, a sensible approach can be to deactivate the *Battery Optimiser* aspect. It is clear that if there was enough available memory, all aspects would run smoothly, so that no conflict would arise.

A list of conflicting situations must be constructed. Each situation must be described as [resource-state] pair list, which must be accompanied with corresponding corrective actions.

How to find those conflicting situations is outside the scope of this paper, and it is part of our related and future work.

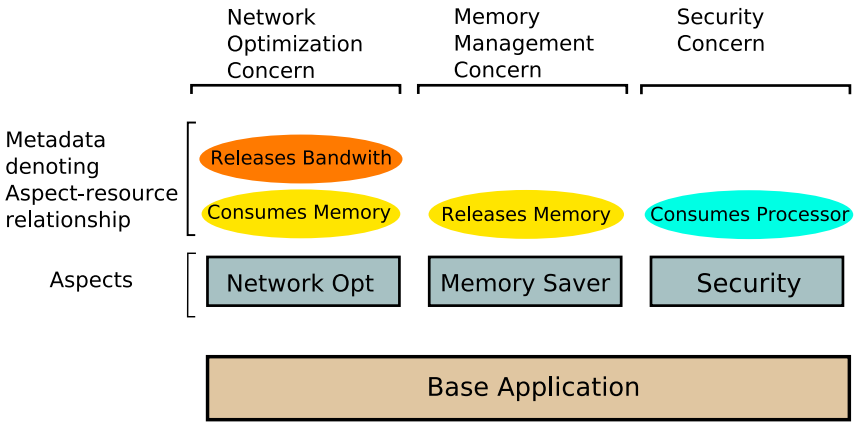

Figure 1. Aspects' metadata indicating effects on resources.

4.2 Solving Conflicts

In this section we present the corner-stones of our vision of conflict resolution.

4.2.1 Semantic Labels

Java annotations [10] (and .Net attributes [13]) are powerful mechanisms that enable lightweight language extensions. They are used with multiple purposes such as attaching domain modelling information and enabling constraint enforcement. More recently, they have been used, along aspects, to demark and modularise crosscuting-cuttings concerns [11]. In order to explicit the use we make of annotations and to differentiate them from other general purpose annotations, we will call them *semantic labels* when they are utilised in the context of this work.

In order to solve/avoid runtime aspectual conflicts we propose the use of semantic labels. Semantic labels are descriptors added to aspects. These descriptors expose aspects' metadata indicating the kind of resources utilised by the aspect and the way they are affected (for instance, consumed or released), that is, the relationship between an aspect and the resources manipulated by it. In other words, semantic labels denote the role played by the aspect regarding a resource.

Figure 1 shows each aspect with their respective metadata, which describes the kind of operations the aspect performs on a resource.

Semantic labels provide a more abstract way of talking about aspects, as we shall see later. In fact, they define a discourse domain for aspects and resources.

4.2.2 Coordinator Aspect

An extra aspect is necessary in order to control the execution of other aspects and solve conflicts. We call this aspect "Coordinator". Semantic labels are consumed by the *Coordinator* aspect in order to have a complete picture of aspects, resources and their relationship.

The *Coordinator* monitors resources' state looking for patterns indicating conflicting situations (Section 4.1), that is, certain state-resource pair patterns. Coordinator is supplied with a set of strategies. Each strategy is associated to a conflicting situation, and is defined as actions to be taken on the aspects involved. For example, a strategy can be defined as *switch off all memory consumers aspects*. Therefore, strategies are expressed as operations on aspects playing defined roles. Furthermore, since roles are used to express strategies, quantification can be achieved.

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

Listing 1. Annotated Aspect

When a conflicting situation is found, the *Coordinator* applies the corresponding strategy. This means that it looks for the aspects playing the roles and performs some management operations on them. By performing the operations, the system behaviour is affected, and the conflicting situation eliminated.

Notice that, for sake of paper's length, we are talking about a coordinator aspect as a single module, when it is actually splitted in several parts.

4.3 Design Details

Figure 2 outlines the design of the proposed solution's prototype. The abstract ResourceHandler aspect provides the basic functionality that allows aspect management, that is switch-on/off behaviour.

Subaspects are defined in order to perform several optimisations on the base program. These aspects must declare which resources they affect and the way it is done (denoted as UML comments [2] in the diagram). Such declarations are performed by annotating the aspects' code, as shown in Code Listing 1. Each annotation refers to a particular resource, and the parameter express how the resource is used.

The Coordinator aspect monitors the evolution of resources states. When a change is detected, a snapshot of resources states is passed to the rule engine. Then, some strategies will be activated according to conflict-prone situations at hand. These active strategies perform corrective actions on the application's aspectual world. They can switch on/off aspects as necessary, so that conflict is neutralised.

[2] Since there is no unified way of expressing annotations in UML we follow one of the possibilities presented in [6]

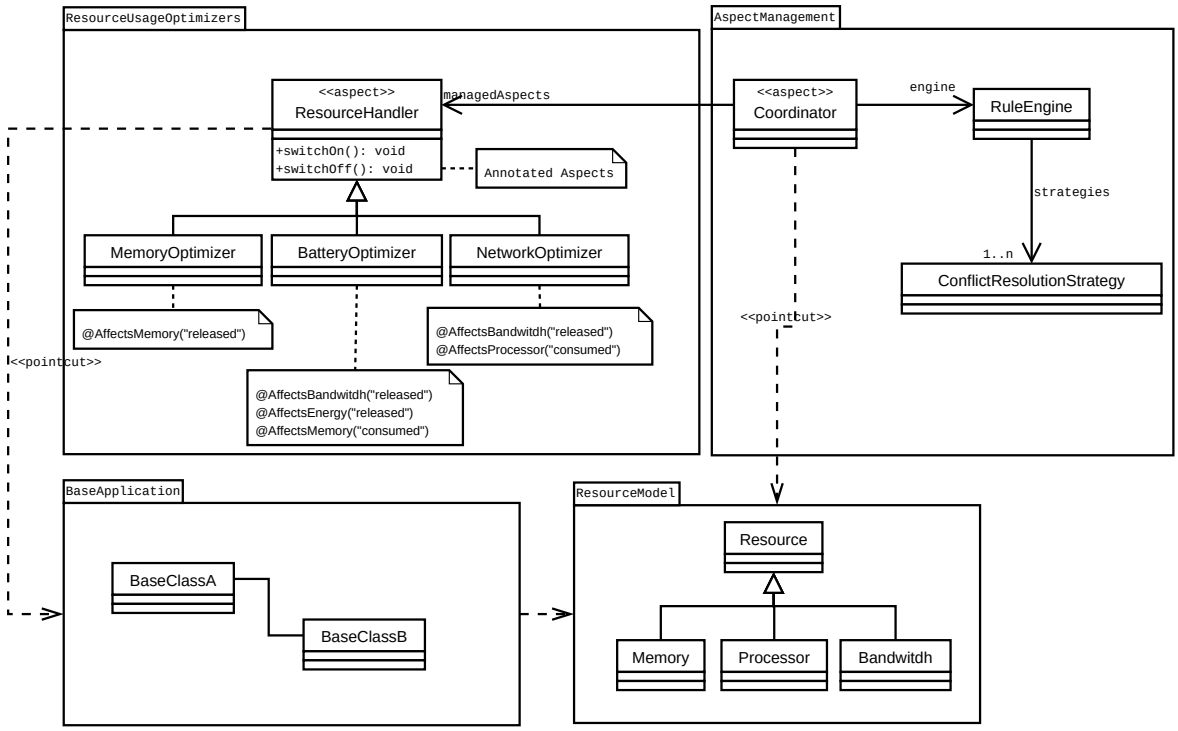

Figure 2. Conflict Resolution Approach Class Diagram.

```
1   rule "ControlMemory"
2    no-loop true
3   when
4    m : Resource(name == "memory",
5                 availability < 10 )
6   then
7    Coordinator.stop("@AffectsMemory
8                      (CONSUMED)")
```

Listing 2. A simple rule for aspect management

Aspects are not directly named by strategies, instead annotations are used to refer to aspects in an abstract form (see code listing 2).

5 Conclusions and Future Work

In this paper it has been shown how semantic conflicts can be solved. The foundations for semantic conflict resolution among aspects have been stated. They include the use of semantic labels for aspects, the early characterisation of conflicting situations, and the use of a coordinator aspect to detect and resolve runtime conflicting situations.

The proposed approach brings the benefits of coordinated aspectual behaviour. At the same time, obliviousness among aspects is preserved by using metadata.

Aspects can be designed and implemented separately and, later on, in an integration phase, their conflicts can be studied and strategies for solving them implemented. Since strategies are expressed in terms of aspects' metadata, strategies are loosely coupled to aspects. Therefore, they can be easily reused. Besides this, strategies define tailored priorisation for aspects in each specific situation, this feature contrasts with other fixed priorisation approaches such as [12].

Unlike other approaches, where a redesign of the aspects is required when a conflict is found [5, 8], this approach allows the independent development of aspects, leaving conflict resolution isolated from the aspects.

Using this approach it is actually possible to prevent conflicting situations by building a set of strategies that carefully depict potential problematic situations and deactivate them before becoming a real problem.

These ideas have been illustrated in the field of resource aware, but they can be extrapolated to other situations where aspects manage a common base of resources.

Our future work includes the generalisation of the presented approach in order to cope with aspectual conflicts in different usage contexts. Part of this work may involve the development of onthologies and the definition of conflict resolution strategies in terms of them.

6 References

[1] Java aspect components. http://jac.objectweb.org/.

[2] Jboss aop. http://www.jboss.org.

[3] Spring framework. http://www.springframework.org.

[4] 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.

[5] 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.

[6] V. Cepa and S. Kloppenburg. Representing explicit attributes in uml. *7th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005.

[7] A. Dantas and P. Borba. Developing adaptive j2me applications using aspectj. *J. UCS*, 9(8):935–955, 2003.

[8] 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.

[9] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, Boston, 2005.

[10] JCP. A metadata facility for the javatm programming language, 2004. http://www.jcp.org/en/jsr/detail?id=175.

[11] R. Laddad. Aop and metadata: A perfect match, March 2005. http://www-128.ibm.com/developerworks/java/library/j-aopwork3/.

[12] A. M. D. Moreira, J. Araújo, and A. Rashid. A concern-oriented requirements engineering model. In O. Pastor and J. F. e Cunha, editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.

[13] MSDN. C # language specification - attribute specification. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec/html/vclrfcsharpspec_17_2.asp.

[14] C. Shin and W. Wook. Conflict resolution method using context history for context-aware applications. In *First International Workshop on Exploiting Context History in Smart Environment. Pervasive 2005*, 2005.

[15] 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.

[16] 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.