Reasoning about Semantic Conflicts between Aspects

Pascal Durr*, Lodewijk Bergmans, Mehmet Aksit

University of Twente, The Netherlands
{durr,bergmans,aksit}@ewi.utwente.nl

Abstract. Aspects have been successfully promoted as a means to improve the modularization of software in the presence of crosscutting concerns. The so-called *aspect interference problem* is considered to be one of the remaining challenges of aspect-oriented software development: when multiple aspects share the same join point, undesired behavior may emerge. Such behavior is not necessarily caused by a wrong implementation of the individual aspects, but may be the result of composition of the independently programmed aspects at the shared join point. This paper presents a language-independent technique to detect semantic conflicts among aspects that are superimposed on the same join point.

1 Introduction

Aspect-Oriented Programming (AOP) aims at improving the modularity of software in the presence of crosscutting concerns. AOP languages allow independently programmed aspects to superimpose [1] behavior at the same join point. Unfortunately, such expression power may cause undesired emerging behavior. This is not necessarily due to a wrong implementation of the individual aspects; the composition of the independently programmed aspects at the shared join point may cause emerging conflicts due to unexpected semantic interactions. Note that interference between aspects may also occur in other places without shared join points, but in this paper we concentrate on this –most relevant– case. In this paper we use the term *semantic* to designate the *behavior* of a component (aspect), rather than its syntax or structure. A *semantic conflict* is emerging behavior that conflicts with the originally intended behavior of one or more of the involved components.

In component-based programming, each component *explicitly* composes its behavior from fine-grained actions and the interfaces of other components. For example, behavior is composed as a sequence of function calls, through the specification of inheritance or through aggregation, e.g. of objects. In all these cases, the programmer is responsible to ensure that the specified composition is sensible. In addition, techniques like type checking support the programmer to avoid certain mistakes, e.g. introducing method with the same name.

* This work has been partially carried out as part of the Ideals project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program.

[1] We prefer to use the term *superimposition* instead of *weaving*, since it less commonly associated, or confused, with implementation techniques.

The reasoning techniques, such as type-systems, that are developed for components cannot be directly applied to aspects at shared join points, because this kind of behavioral composition is *implicit*: each aspect is defined independently of the others, potentially at different times and by different people. The composition of their advice happens 'by coincidence' at shared join points, certainly the programmers of the individual aspects cannot always be aware that this will happen.

Recently, reasoning about the correctness of a system after superimposing multiple aspects at the same or shared join point, as described in [20], has been considered as an important problem to be addressed[16,17,12]. Our approach focuses on semantic conflicts, not conflicts that are syntactic or structural, for example changing the inheritance hierarchy while another aspect depends on the original hierarchy.

This paper presents a language-independent technique to detect emerging semantic conflicts among aspects that share join points. The paper is structured as follows; in section 2 we explain the problem statement through a simple example of a semantic conflict, based on a system with an Encryption and a Logging aspect. Subsequently, section 3 provides an overview of our approach. Finally we provide an overview of related work in section 4 and conclude.

2 Problem statement

To illustrate the kinds of conflicts we consider, we present an example with two crosscutting concerns. One may of course discover numerous other examples of semantic conflicts between aspects, see for example[10].

Consider a base application which implements a simple protocol. Here, to handle inbound and outbound messages, the interface of class Protocol provides the methods sendData and receiveData. Now let us assume that we would like to add two new aspects: logging and encryption. LoggingAspect is applied to the join points where the methods sendData(String) and receiveData(String) start execution. This aspect prints the arguments of both methods. EncryptionAspect [19] provides encryption functionality for all outbound messages and decryption for all inbound messages. The base system with both aspects is shown in figure 1.

In this example, both the logging advice and the encryption advice are applied to the same method sendData(String). Similarly, the logging advice and the decryption advice are applied to the same method receiveData(String). These two join points create semantic interference, as we will discuss in this section.

Consider, as an example, the method sendData(String). Now assume that the logging aspect is used for debugging purposes and should be applied to non-encrypted messages only. In this case, it seems to be a logical option to apply the logging advice before the encryption advice. However, one could also argue that the reverse order is preferable in a "hostile" context where the messages must be encrypted first before sending them to the debugger. The exact order must be determined based on the requirements of the domain or even the individual application and therefore cannot be determined in a generalized automated manner. We assume that it is required to apply logging before encryption. In this case, we consider applying the logging aspect *after* encryption as an undesired interference (i.e. a conflict) between these two aspects.

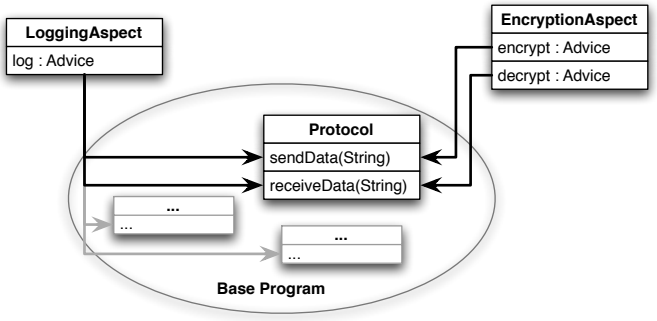

Fig. 1. Encryption and Logging example

Similarly, it is also possible to identify a similar conflict for the method receiveData(String). At this method, the decryption aspect must be applied first before the logging advice. The reverse order is in this example considered as another semantic conflict.

Now let us elaborate more on these two conflicting aspects. Individually, both aspects are consistent with their requirements and therefore they are considered sound. From the language compiler point of view, the program with the conflicting order of advices is considered as a valid program without error(s). However, once these aspects are applied at the same join point, an emerging conflict situation appears. Such a semantic conflict may lead to undesired behavior.

In this case, if one is aware of such (potentially) conflicting cases, he/she can enforce an ordering. For example, it is possible to enforce an ordering in AspectJ with the declare precedence construct. In practice, however, detecting emerging conflicts may not be that easy, especially when conflicting aspects crosscut the entire base application and share many join points. It is therefore necessary to develop techniques and tools that reason about the (potential) semantic conflicts between aspects.

3 Approach

To reason about the behavior of advices and detect semantic conflicts between them, we need to introduce a formalization that enables us to express behavior and conflict detection rules. Clearly, a formalization of the complete behavior of advice in general would be too complicated to reason with. Therefore, an appropriate abstraction must be designed that can both represent the essential behavior of advice, and be used to detect semantic conflicts between advices.

Our approach is based on a resource-operation model, also called resource model, to represent the relevant semantics of advice, and detect conflicts among them. We have chosen to adopt a resource model, as this is an easy to use model that can represent both very concrete, low-level, semantics and very high-level, abstract behavior. For more detailed information about the model and its usage we refer to [21]. Our approach of

conflict detection resembles the Bernstein[3] sufficient condition's for determinacy. A similar approach is also used for detecting and resolving conflicts in transaction systems, such as databases[18].

The key idea is that some resource must be shared among advices for the advices to conflict. Hence semantic conflicts can be represented by modeling the operations that advice performs on some shared resource. In the following we will explain the model intuitively, based on the previously presented example. In [9] and [21] we present this model more formally and provide a concrete instantiation of the model for the Composition Filters[2] approach and an implementation in Compose*[25]. To summarize: our semantic specification language is a resource model which is attached to the advice.

Figure 2 presents the semantic analysis process and the relationships to the base system and advice. We use this image as a guideline through sections 3.1, 3.2 and 3.3.

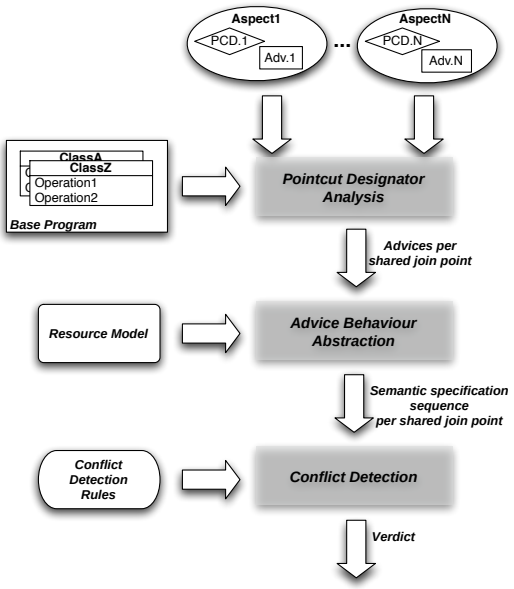

Fig. 2. Overview approach

3.1 Pointcut designator analysis

At the top of the figure a set of aspects (Aspect1 ... AspectN) is presented. These aspects contain advices and Pointcut Designators (PCD). There is also a base system with a set of classes (ClassA ... ClassZ). The aspects and base are inputs of the *Pointcut Designator Analysis* phase. During this phase all PCDs are evaluated with respect to the base program. This results in a set of join points with advice(s) superimposed on them. Our approach only considers the join points with multiple advices superimposed on them, these are also referred to as a shared join points.

3.2 Abstraction

After the PCD analysis phase, we retain a sequence of advices per shared join point[2]. This sequence is used in the *Abstraction* phase. The other input for this phase is the resource model. During the *Abstraction* phase, the sequence of advices are transformed into sequences of resource-operation tuples per shared join point. We now discuss the notion of Resources and Operations and provide instantiations of these notions for the running example.

Resources A resource is in essence an Abstract Data Type[13]; its identity is determined by the operations carried out on it. A resource has an alphabet, a set of operations which are allowed to operate on this resource. One such resource, in our example, is called arguments, which represents the arguments of received or sent messages. In fact, the logging, encryption and decryption advices all operate on a resource arguments. For example, the logging advice reads the argument of a message, whereas the encryption advice modifies the same argument. Similarly, the decryption advice affects the resource arguments. The arguments resource is a one example of a resource, other examples are; a lock, a buffer or the return value of a method.

Operations The logging advice accesses the arguments, this is a read operation on the arguments resource. The encryption advice encrypts the same arguments resource. Similarly, the decryption advice also operates, with a decrypt operation, on the arguments resource.

Although the very primitive actions on shared resources are basically read and write operations, if desired by the programmer, we think that such actions must be modeled at a higher level of abstraction. For example, in this paper, we will model both encryption and decryption advices as respectively encrypt and decrypt operations instead of read-write operations.

There is a subtle difference between changing the content of the arguments and transforming or encapsulating the data, as is the case with encryption. The intended meaning of the encryption and decryption advice is not to change the arguments. Also we would have lost our ability to distinguish between two, semantically, different actions: encryption and decryption. We chose to model the log action as a read operation as this adheres to the intended meaning of the advice. In short, the programmer must be able to choose his/her own higher-level operation definitions on the shared resources instead of primitive read-write operations only.

3.3 Conflict detection

The operation sequences per resource per shared join point are, in combination with the conflict detection rules, the inputs for the *Conflict Detection* phase. This phase passes a verdict, i.e. if a conflict is present or not, for each shared join point and for each combined sequence of operation per resource.

[2] In the case that the ordering is partially known, we iterate over the *Advice Behavior Abstraction* and *Conflict Detection* phase for each valid ordering.

Conflict detection rules A conflict detection rule is a requirement on a resource. This is specified as a matching expression on the sequences of operations per resource. This rule can either be an assertion pattern, a combination of operations that must occur on a resource, or as a conflict pattern, a combination of operations that must not occur.

In the example used in this paper, a conflict situation is specified as: "if a read operation occurs after an encrypt operation on the same resource, then it is considered as a conflict". Another conflict rule is specified as: "if a read operation occurs before a decrypt operation on the same resource, then it is considered as a conflict". This can be expressed with a matching language, such as temporal logic, regular expressions or predicate based. The conflict rules are specified for the domain of the resource it constraints, the can thus be broader than a specific application. Only the conflict rules for application specific resources cannot be reused beyond that specific application.

For instance, we can formulate these two requirements, on the arguments resource, as the following conflict detection rule: $(encrypt\ before\ read \mid read\ before\ decrypt)$.

In case of detecting an error, several actions can be carried out, such as reporting the conflict to the programmer or aborting the compilation process.

Conflict analysis For each shared join point, there is one sequence of operations on the resource arguments. In our example, we thus have two sequences, one for the join point sendData(String) and one for join point receiveData(String). Now assume that first an encrypt and then a read operation (caused by the logging concern) occur on the arguments resource at a shared join point. This would match the conflict detection rule: $(encrypt\ before\ read \mid read\ before\ decrypt)$, in which case the verdict of the conflict detection process is: "conflict".

4 Related work

In [15,14], Katz proposes three categories of aspects: spectative, regulative and invasive. The spectative aspects do not influence the underlying system, they only query the state of fields. Regulative aspects can alter the control flow of the underlying system. Finally, invasive aspects both alter the control flow and the fields of the underlying system. They are able to determine whether the combination of these aspects interfere with one another. We can also classify aspects by inspecting the ResourceUsageMaps of the advices and determine if they change the control flow, e.g. by writing the target, or if they are spectative, e.g. only reading resources. Clifton and Leavens [4], also propose a classification system based on observers and assistants. Again this classification can be achieved by constraining the usage of resource operations. Our approach offers a more fine-grained interference detection mechanism, that the classifications described previously. There is also the issue that although they might know that aspect interfere they are unable to state whether this interference is undesired.

Rinard, Salcianu and Bugrara[24] also propose a classification system for advices. Their categories are: *Augmentation*, *Narrowing*, *Replacement* and *Combination*. They classify interactions between the advice and the base code in terms of the usage of the same fields. Their approach is based on the fact that if the advice interferes with base system that there should be some shared field. This is similar to our resource

definition. They distinguish two operations on these fields, read and write. They use code analysis tools to determine which operations the advice and base systems do. Based on the interaction analysis on similar fields they define they following types of interactions: *Orthogonal*, *Independent*, *Observation*, *Actuation* and *Inference*. A similar classification can also be made on the bases of our resource model. Similar to the slicing technique discussed below, they are unable to indicate whether such an inter action is desired or conflicting. Furthermore they do not allow the use of abstract resources, which can capture more subtle problems that accessing similar fields.

Douence, Fradet and Sudholt[6][5][23][8][7] present a technique to detect shared join points, based on similarities in the crosscut specification of the aspects involved. If there is no conflict the aspects can be woven without modification, else the user has to specify the order in which the aspects should be composed. They do not consider the semantics of the advice on inserts, they consider the presence of a shared join point to be a conflict.

Program slicing techniques as presented by Balzarotti, Casteldo and Monga[1] also provide aspect interference detection. They propose an approach for slicing AspectJ woven code. The detection is based on checking whether the nodes of one aspect slice appear in the slice of another aspect; if this is the case, there is interaction between the aspects. They are all able to detect possible interference, e.g. read-write conflicts, and even pinpoint the exact memory location. The main drawback of using slicing tools is that there is no way to know if the interaction is desired or not. These are generic based conflicts, it is hard to take application specific conflicts into account. They do have the advantage of, automatically, being able to determine conflicts due to side effects of advices.

In [22], Pawlak, Duchien and Seinturier present a language called *CompAr*, which allows the programmer to specify the execution constraints of the advice. And it provides an abstraction from the implementation language. This technique also analyzes the issues found at shared join points. The *CompAr* compiler verifies whether the execution constraints hold for that given abstract specification. The work focuses on determining the correct order of composition given the execution constraints. They do provide a means to express certain actions that have to be carried out, but there is no interference detection between these actions, as is the case in our approach.

5 Conclusion

This paper presents a novel approach for detecting semantic conflicts between aspects. Our approach defines the semantics of advice in terms of operations on a resource model. After analyzing all advices at a shared join point, we are able to detect conflicts based on conflict patterns over the combinations of operations on these resources. The resource-operation model allows us to express knowledge about the behavior of advice at both concrete and abstract levels.

The presented approach is generic and can be applied to most, if not all, AOP languages. It requires the ability to detect shared join points for such a language and the ability to annotate advice with the semantic resource operation specifications. In that

case our model supports reasoning about the behavior of the composition of multiple advices.

Our conflict detection model is generic and abstract. We imagine that we can apply the same conflict detection approach on a higher design level, i.e. at the requirements or architectural level. As the resource model is generic enough, we can use our approach for these cases and even reuse parts of the Compose* toolset for this. This is also considered future work.

has been implemented in Compose*

In the composition filters approach we exploit the declarative specification of filters to automatically derive the semantics from the filter specifications. As input to this process serve the specifications of the predefined filter types, and possibly the annotations of user-defined advice (through meta-filters). The paper also discusses our implementation of this approach within the Compose* toolset.

We believe the approach presented in this paper offers a powerful and practical means of establishing semantic conflict detection with a minimal amount of explicit behavior specifications from the programmer.

References

1. D. Balzarotti, A. Castaldo, and M. Monga. Slicing aspectj woven code. In *FOAL '05: The 4th Workshop on Foundations of Aspect-Oriented Languages*, Chicago, USA, March, 14 2005.
2. L. Bergmans and M. Akşit. Principles and design rationale of composition filters. In Filman et al. [11], pages 63–95.
3. A. J. Bernstein. Program analysis for parallel processing. *IEEE Trans. on Electronic Computers*, EC-15:757–762, 1966.
4. C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In R. Cytron and G. T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 33–44, Mar. 2002.
5. R. Douence, P. Fradet, and M. Südholt. Detection and resolution of aspect interactions. Technical Report RR-4435, INRIA, Apr. 2002.
6. 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.
7. R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In Filman et al. [11], pages 201–217.
8. R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Segura-Devillechaise, and M. Sudholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.
9. P. Durr, L. Bergmans, and M. Aksit. Technical report: Formal model for secret. Technical report, University of Twente, 2005.
10. P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS '05: The 2nd European Interactive Workshop on Aspects in Software*, Brussel, Belgium, September, 1-2 2005.
11. R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
12. J. Hannemann, R. Chitchyan, and A. Rashid. Analysis of aspect-oriented software, workshop report. In *ECOOP 2003 Workshop Reader*, Darmstadt, Germany, July, 21 2003.

13. D. Kapur and S. Mandayam. Expressiveness of the operation set of a data abstraction. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 139–153, New York, NY, USA, 1980. ACM Press.
14. S. Katz. A superimposition control construct for distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(2):337–356, 1993.
15. S. Katz and J. Gil. Aspects and superimpositions. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 308–309, London, UK, 1999. Springer-Verlag.
16. G. T. Leavens and C. Clifton. Foundations of aspect-oriented languages workshop. In *Foundations of Aspect-Oriented Languages Workshop*, volume 3rd. AOSD, 2004.
17. G. T. Leavens and C. Clifton. Foundations of aspect-oriented languages workshop. In *Foundations of Aspect-Oriented Languages Workshop*, volume 4th. AOSD, 2005.
18. N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete. *Atomic Transactions : In Concurrent and Distributed Systems*. Morgan Kaufmann, 1993.
19. L. Z. Minwell Huang, Chunlei Wang. Toward a reusable and generic security aspect library. In *AOSD:AOSDSEC '04: AOSD Technology for Application-level Security*, Lancaster, UK, March, 23 2004.
20. I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *Proceedings of International Conference NetObjectDays, NODe2005*, Lecture Notes in Computer Science, Erfurt, Gergmany, 2005. Springer-Verglag.
21. Pascal Durr. Detecting Semantic Conflicts Between Aspects, 2004. http://www.cs.utwente.nl/~durr/papers/ Master_Thesis_Pascal_Durr.pdf.
22. R. Pawlak, L. Duchien, and L. Seinturier. Compar: Ensuring safe around advice composition. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems*, Athens, Greece, June 2005.
23. M. S. Remi Douence, Pascal Fradet. A framework for the detection and resolution of aspect interactions. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, Lecture Notes in Computer Science, Pittsburgh, US, October,6 2002. Springer-Verglag.
24. M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FOSE)*. ACM, Oct. 2004.
25. University of Twente. COMPOSE*. http://composestar.sourceforge.net.