# Safe Aspect Composition

Laurent Bussard[1], Lee Carver[2], Erik Ernst[3], Matthias Jung[4],
Martin Robillard[5], and Andreas Speck[6]

[1] I3S – CNRS UPRESA 6070 – Bât ESSI, 06190 Sophia-Antipolis, France, bussard@essi.fr
[2] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, lcarver@us.ibm.com
[3] Dept. of Computer Science, University of Aalborg, Denmark, eernst@cs.auc.dk
[4] Institut Eurécom, 06190 Sophia Antipolis, France, jung@eurecom.fr
[5] Dept. of Computer Science, University of British Columbia, Canada, mrobilla@cs.ubc.ca
[6] Wilhelm-Schickard-Institute for Comp. Sci., U. of Tübingen, Germany,
specku@informatik.uni-tuebingen.de

## 1 Introduction

When different aspects [4] are composed, one must ensure that the resulting composition does not cause conflicts. This problem appeared to be central to the work of the authors. The corresponding discussions that took place during the workshop led to a categorization of conflicts relating to the composition of aspects.

We thus present three general classes of conflicts associated with aspect composition, and consider possible remedies. The first kind of conflicts, discussed in Sec. 2, is *inherent*, i.e., it is the kind of conflicts where a given combination of aspects should be rejected by the translation system. Section 3 deals with the situation where certain aspects should be combined but cannot—because of *accidentally* conflicting characteristics of the implementation. Finally, as described in Sec.4, aspect combinations may give rise to *spurious* conflicts, where the combination is intended to work—and it would actually work at run-time—but the type check fails. In these cases we may have to choose between loss of reuse opportunities or loss of type-safety, or we must use a more powerful type analysis.

Generally, name resolution plays an important role in the emergence and handling of composition conflicts. The various aspect composition technologies necessarily relax the conventional name resolution rules. The inclusion of an aspect or a concern relies on a tolerant form of name matching that precludes detection of overdefined or underdefined names. Current tools, such as Hyper/J [6] and AspectJ [4], provide no mechanisms to prevent or require the inclusion of specific aspects or enhancements. This issue must be addressed in order to handle the technical details of our challenge problems.

## 2 Inherent Conflicts

Assume that we have a set of aspects available, such that many combinations of them are appropriate but others should be avoided. The following example demonstrates this. In this example, aspects are used to separate distribution

related code (CORBA code) from problem-domain related code. This reduces the complexity and improves the reusability of the code.

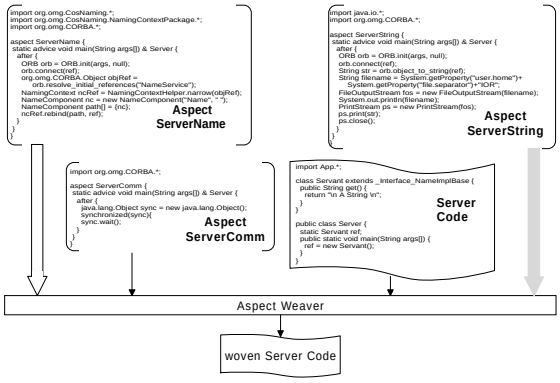


**Fig. 1.** CORBA server example: use exactly one of `ServerName`, `ServerString`

Figure 1 shows a simplified version of the example. Details may be found in [5]. In the example, clients will send requests to a server and this server will respond with a string. The implementation consists of four units—the *Server Code*, and the aspects `ServerName`, `ServerString`, and `ServerComm`. The *Server Code* implements the server which manages client requests. The aspects encapsulate all CORBA-related code.

To use a service, the client has to obtain an initial reference to the server. This can be done either via a name server or via a disk file, but only one of these methods can be chosen. `ServerName` encapsulates the name server based solution, while `ServerString` realizes the file based solution. As a result, we must apply exactly one of these aspects.

`ServerName` and `ServerString` could possibly both be applied in the same server without compile-time errors, but the result would be a server with incorrect behavior. The conflict is *inherent*, and the correct response would be for the composition mechanism to reject the combination. To handle this, it should be possible to write specifications of aspect compatibility, which would then be checked at composition-time. Hyper/J [6] could easily be extended to express mutual exclusion by putting the two aspects in the same dimension and specifying that only one coordinate can be selected from that dimension. An automatic detection of all semantic conflicts would of course be undecidable, so we must support manual annotations expressing the combination constraints.

## 3 Accidental Conflicts

Sometimes it is intended to be possible to compose a certain set of aspects, but the composition still fails for accidental reasons. In these cases the desired course

of action is not just to detect and reject the composition—even though detection of the problem is still the first step. The conflict should be *resolved* by somehow adjusting the aspects or their composition. This will become a more and more common and serious problem when aspects get to be reused in different contexts, and aspects written by different people get to be used in the same context.

Let us consider an example of such a conflict, presented in detail in [2]. In the example, the core program contains a bank `Account` class, and there are two aspects `Event` and `Transaction` providing extra features to the `Account` class.

The `Event` aspect adds event handling to `Account`, so an `Account` will transmit events on certain channels whenever a significant transition occurs in the state of the `Account`. Other objects may then listen on those channels, thus being notified about the development. The `Transaction` aspect makes it possible for the `Account` to support transaction semantics via the operations `begin`, `commit`, and `rollback`.

When both aspects are used together, a semantic conflict arises: Imagine that a client performs a `deposit` operation on an `Account` object, and this operation is terminated by a `rollback`. Now, the `Event` aspect might have caused broadcasting of events to unknown listeners during the execution of `deposit`. The `rollback` will leave the `Account` itself in the correct state (as if nothing happened), but the event listeners will have the impression that the `deposit` actually took place—events during the aborted operation cannot be retracted.

In order to handle this composition of aspects, it should be possible to delay the actual event broadcasting until the `deposit` method finishes and either `commit` or `rollback` is executed—the events would then be released in the former case and suppressed in the latter case. However, the problem must first be detected, and the resolution is not just a syntactic composition of the aspects. One possible solution would be to use composition filters [1] to intercept all the event transmissions and store the events until the end of the method—but even then it would not be simple to make the two aspects work together seamlessly. Another approach, using an extended Hyper/J, would be to require that the `Event` aspect must provide certain transaction related services, thus making it possible for the `Transaction` aspect to integrate the `Event` aspect into the transactional environment.

## 4 Spurious Conflicts

Until now we have discussed actual conflicts, handled by rejecting the program or resolving the conflict. In this section we discuss the opposite situation, where the composition mechanism reports a conflict even though there is really no conflict.

To illustrate the point we look at an analogous situation. C++ was once implemented by means of a source-to-source compiler called Cfront. Cfront translated C++ code to C code, which was then compiled by an ordinary C compiler. Cfront performed a full static analysis of the C++ program, and it was considered a bug in Cfront if the back-end C compiler ever reported an error.

Now consider how this would have worked if Cfront had not done any static analysis, but instead relied on the C compiler to indirectly check the C++ program by checking the translated C code. Consider the following simple program:

```
class Point {...};
class ColorPoint: public Point {...};
int main(...) { Point * pp = new ColorPoint(); ... }
```

In this example, the pointer `pp` has type `Point` and it will point to an instance of `ColorPoint`. This is just ordinary subtype polymorphism, but the point is that there is no support for polymorphism in C, so there would have to be an unsafe type cast in the translated C code.

In other words, if C++ type checking is left to a C compiler, subtype polymorphic pointer assignments become indistinguishable from arbitrary, unsafe pointer assignments. We would have only two options: prohibiting subtype polymorphism and thereby losing reuse opportunities (and much more than that), or accepting loss of type safety.

The same situation arises with aspect composition mechanisms based on textual transformations: the generated code sometimes seems to be unsafe even though it is type safe, because the type analysis in the target language, e.g. Java, is not sufficiently powerful to capture the concepts expressed in the aspect language.

An example of this is given in Sec. 4 of [3]. An outline of this example is as follows: There is a family of classes which is used to build expressions: an abstract class `Expression` and concrete subclasses `Number` and `Plus`. Along with this family comes a family of visitors: abstract `Visitor` and concrete `Evaluate` and `Show`. The visitors are used as specified in the *visitor* design pattern and they make it possible to evaluate and print a given expression.

Now the conflict arises if a system contains more than one instance of these families[1] and they should be used in a polymorphic manner, i.e., if there is a need to be able to visit different kinds of expressions polymorphically. The problem is that the different families of classes will either be unrelated according to the type system, or they will be indistinguishable. This means that polymorphic code will either be rejected—and this is the spurious conflict that this section is all about—or we will have to give up type safety. The solution to this problem is to use a more expressive type analysis, such as that in [3]—i.e., the solution is to treat the language with aspects as a language in its own right, and then perform full static analysis at that level.

## References

1. Aksit, M., and Tekinerdogan, B. Aspect-oriented Programming Using Composition Filters. In Demeyer, S. and Bosch, J. (Eds) *Object-Oriented Technology, ECOOP'98 Workshop Reader*. Springer-Verlag, 1998.

[1] Current aspect systems do not allow one program to contain more than one aspectualization of any given class, but it would be a natural development to make this possible.

2. Laurent Bussard. *Towards a Pragmatic Composition Model of CORBA Services Based on AspectJ* Position paper accepted at the workshop on Aspects and Dimensions of Concern at ECOOP'2000. Cannes, FRANCE, June 2000.
3. Erik Ernst *Separation of Concerns and Then What?* Position paper accepted at the workshop on Aspects and Dimensions of Concern at ECOOP'2000. Cannes, FRANCE, June 2000.
4. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Lecture Notes in Computer Science LNCS 1241*, ECOOP. Springer-Verlag, June 1997.
5. E. Pulvermüller, H. Klaeren, and A. Speck. Aspects in Distributed Environments. In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering GCSE'99*, Erfurt, Germany, September 1999.
6. Harold Ossher, Peri Tarr. *Multi-dimensional Separation of Concerns in Hyperspace.* Position paper at ECOOP'99 Workshop on Aspect Oriented Programming. In *Lecture Notes in Computer Science LNCS 1743*, ECOOP. Springer-Verlag, 1999.