

# Integrating ACI into the Aspect Sandbox

Conrad Mueller

December 03, 2003

## 1 Introduction

In this paper, I build on work done by Hidehiko Masuhara and Gregor Kiczales [1] on a unified modeling framework for different aspect-oriented systems. I show that Aspect Collaboration Interfaces (ACI), as proposed by Mira Mezini and Klaus Ostermann in [2] and being implemented in the *CAESAR* system, can be integrated into that framework.

## 2 About the modeling framework

The framework proposed by Masuhara and Kiczales [1] “models each aspect-oriented mechanism as a *weaver* that combines two input programs to produce either a program or a computation.” I use the terms and abbreviations of the framework throughout this paper.

As a part of their work, they have implemented an interpreter for a simple object-oriented language with scheme-like syntax in scheme (the *Aspect Sandbox*). They use this interpreter as an environment and a basis to implement multiple different aspect-oriented mechanisms.

These implementations are in turn used to intuitively “prove” the correctness of the modeling of each mechanism in terms of the framework. I present a similar argument in section 5.

For more information regarding the framework, the reader is referred to [1].

## 3 About ACI

The ACI mechanism divides aspects into two parts, *implementation* and *binding*.

Conceptually, the implementation is the more technical part of the aspect. Examples are algorithms on generic data structures like trees, or specific implementations of aspect infrastructure (i.e. a back-end for the observer pattern where each subject has a linked list of its observers). These implementations are generic and can thus be reused in the context of multiple different programs.

The binding on the other hand re-modularizes the rest of the system in terms of the aspect by mapping classes present in the system to “roles” defined in the

ACI. Simple examples are just wrappers for existing classes, while more complex bindings could map the existing data structures to others (like, for instance, a generic tree structure) on the fly. Bindings are specific for the context of a given program, but can be reused with multiple different implementations (for example, to apply different tree algorithms to the rest of the system)..

To achieve this, an ACI is an interface that can have three kinds of members: nested ACIs, methods marked as *provided* and methods marked as *expected*.

A class that is marked as an implementation of an ACI has to provide an implementation for each *provided* method and a nested class for each sub-interface of the ACI that contains a *provided* method.

Similarly, a class that is marked as a binding of an ACI has to provide an implementation for each *expected* method and at least one nested class marked as a binding of each sub-interface of the ACI that contains an *expected* method.

*Weavelets* combine an implementation and a binding to a complete aspect that can be deployed using *deployment* statements in a dynamic or static fashion.

In [2], Mezini and Ostermann argue that this mechanism allows for better aspect modularity and reuse than the PA mechanism alone. For more information and details, the reader is referred to that paper.

Information about the CAESAR project that aims at implementing ACI as an extension of the Java programming language, as well as information about ACI in general can be found at <http://www.caesarj.org>.

## 4 Modeling ACI in terms of the framework

While I present the results (in this section) and the “proof” (in section 5) in a neat and separate fashion, the process of arriving at both has been much less straight-forward (see also section 7.2)

### 4.1 Parts I have left out

The CAESAR system as described in [2] has multiple advanced features. The most prominent of these features is the ACI mechanism itself, but it also provides a sophisticated *wrapper mechanism* that simplifies the most common case for bindings. Since the wrappers are mostly syntactic sugar and not an integral part of the ACI mechanism itself, I will leave them out of my considerations.

CAESAR also enables *dynamic deployment* of aspects (basically meaning that the advice contained in an aspect can be activated and deactivated at runtime). I will concentrate on *static deployment* of aspects. In section 6, I will present an argument that I would be able to add *dynamic deployment* to the model in a straight-forward fashion.

## 4.2 Rationale of the modeling

The main point of ACI is that it splits aspect definitions into two separate parts, *implementation* and *binding*. A third, separate mechanism that affects the weaving process is aspect *deployment*. That means that, combined with a base object-oriented language, the ACI mechanism consists of four separate languages, while the modeling framework only supports three such languages ('A', 'B' and 'META').

In my view, however, the ACI mechanism merges partial class definitions into complete class definitions. Therefore, the weaver does not need the whole program as input nor output. Rather, it can be invoked whenever weaving is actually needed (that is, for example, on deployment).

This leads to a modeling that somewhat resembles the view on TRAV in [1]. TRAV is embedded in a complete program, only providing parts of its execution ("execution of traversal").

Similarly, an ACI weaver does not produce a complete program or execution as an output, but one weaved class declaration (this is the result 'X' in the model). Since the weaver takes two partial implementation and merges them into a complete one, the join points in 'X' are the elements of these implementations: class and method declarations. Normally, method declarations probably should not be merged at all, but at least constructors would be an exception to this rule, so I include them in this list.

Since conceptually, the aspect implementation provides most of the aspect's behavior, while the binding re-modularizes the surrounding program, I consider implementation to be more similar to the structure of the weaved class. Therefore, I model implementation as the 'A' language and binding as the 'B' language.

Both 'A' and 'B' consist of partial implementations of the final aspect. Base language declarations can be contained in both, while *provided* and *expected* methods are only allowed in the implementation and binding parts, respectively.

When joining the classes together, the matching is based on names, so the methods to identify the join points (' $A_{ID}$ ' and ' $B_{ID}$ ') are the names of classes and methods.

The effect on the join points that can be expressed in the two languages (' $A_{EFF}$ ' and ' $B_{EFF}$ ') is to actually provide the declarations that are to be merged.

Since Weavelets and deployment affect the weaving process, they are identified as the 'META' language.

The ACI declarations themselves do not participate in the weaving; rather, they provide means to enable checking mechanisms and type safety. Thus, they do not show up in the modeling I present here.

### 4.3 The model

In summary, the model of ACI in terms of the framework is as follows:

$X$	Weaved class declaration
$X_{JP}$	Class and method declarations
$A$	ACI implementation (classes, fields and <i>provided</i> methods)
$A_{ID}$	Class and method names
$A_{EFF}$	Provide declarations
$B$	ACI bindings (classes, fields and <i>expected</i> methods)
$B_{ID}$	Class and method names
$B_{EFF}$	Provide declarations
$META$	Weavelets and Aspect deployment

## 5 Implementing ACI

The base language interpreter is patched to call the following function for each deploy declaration on startup:

```
(define ACI:weave-one
  (lambda (deploy-decl)
    (let* ((aciname (deploy-decl-aciname deploy-decl))
           (dname (deploy-decl-dname deploy-decl))
           (implname (deploy-decl-implname deploy-decl))
           (bindname (deploy-decl-bindname deploy-decl))
           (acidecl
            (find-if-else-error
             *aci-decls*
             (lambda (decl)
               (eq? (aci-decl-aciname decl) aciname))
             aciname))
           (impldecl
            (find-if-else-error
             *impl-decls*
             (lambda (decl)
               (eq? (class-decl-cname decl) implname)) ;; ID-A
             implname))
           (binddecl
            (find-if-else-error
             *bind-decls*
             (lambda (decl)
               (eq? (class-decl-cname decl) bindname)) ;; ID-B
             bindname)))
      (if (ACI:check acidecl impldecl binddecl)
          (ACI:weave impldecl binddecl deploy-decl)
          (error 'ACI:weave-one
                 "Error deploying ACI -- ~s" deploy-decl))))))
```

This looks up the ACI, implementation and binding that correspond to the deployment at hand. Data in the global variables (for example, `*aci-decls*`) is also collected on startup of the interpreter.

If a call to a check function (`ACI:check`, which does not yet detect any errors in its current implementation) is successful, the declarations are weaved using the following function:

```
(define ACI:weave
;   A      B      META
  (lambda (impldecl binddecl deploydecl)
    (let* ((dname (deploy-decl-dname deploydecl))
           (weaved-decl (ACI:merge dname impldecl binddecl))
           (finished-decl
            (replace-cname
             (deploy-decl-bindname deploydecl) dname
             (replace-cname
              (deploy-decl-implname deploydecl) dname weaved-decl))))
      ;; activate advice contained in the ACI
      (set! *advice-decls*
            (append *advice-decls* (collect-advice-decls finished-decl)))
      finished-decl)))
```

The two classes along with their nested classes are merged by `ACI:merge` (see below), and all references to the names of the merged classes are updated to refer to the resulting class (its class name is determined by the deployment declaration).

Also, the advice contained in the resulting class are activated.

```
(define ACI:merge
  (lambda (cname impldecl binddecl)
    (let* ((impl-decls (class-decl-decls impldecl)) ; EFF-A
           (bind-decls (class-decl-decls binddecl)) ; EFF-B
           (nested-impl-decls (collect-if class-decl-impl? impl-decls))
           (nested-bind-decls (collect-if class-decl-bind? bind-decls))
           (base-decls
            (ACI:merge-basedecls
             cname
             (remove-all impl-decls nested-impl-decls)
             (remove-all bind-decls nested-bind-decls)))
           (matches
            (ACI:findmatches nested-impl-decls nested-bind-decls))

           (merged-matches
            (map (lambda (match)
                  (map (lambda (bind-decl)
                        (ACI:merge (class-decl-cname bind-decl)
                                   (car match) bind-decl))
                      (cdr match))))
            matches))
```

```
(decls (append base-decls (apply append merged-matches)))  
  
(make-class-decl cname (class-decl-sname bindecl) decls)))
```

This function merges all “base declarations” in the two classes, that is methods and fields. Then it matches existing nested classes and applies itself to each of them in turn. The results are combined into a newly created class declaration.

ACI:findmatches and ACI:merge-basedecls match and merge declarations based on names (like already seen in ACI:weave-one, above). I have not included the source code of said functions here for several reasons:

First, it is not very interesting. Most of these functions consist of matching by name and making sure that nothing gets lost or used twice in the process. These are but implementation details rather than relevant for the model.

Also, and equally important, the merging does not take care of problems like merging of constructors. Since both the implementation and binding classes could inherit from other classes, the resulting class would have to inherit from both in some way. The paper about CAESAR [2] does not talk about these problems.

For these reasons, I have left them out (as well as several other helper functions).

## 6 Dynamic Deployment

While my implementation does not yet enable dynamic deployment of aspects, an implementation of that feature should be relatively straight-forward. The system already activates the advice on deployment. The only additions needed would be a *deploy* expression primitive and a facility to deactivate the advice when the deployment block is left.

I believe that my current implementation facilitates such an extension.

One reason why it does not yet include dynamic deployment is that I concentrating on finding the right modeling for the rest of the features first before implementing every aspect of ACI.

I also believe that dynamic deployment would not change the modeling in any way: it would just add another trigger for the weaving process, while the weaver itself would remain unchanged.

## 7 Discussion

### 7.1 Problems with the base language

ACI concepts are fundamentally based on nested classes and interfaces. The original base language interpreter contained in the Aspect Sandbox does not provide support for that.

I have implemented some support, but nested classes need to be addressed with their fully qualified class name. To enable shorthand notations, the class lookup code would have to be aware of the context of the lookup, which is not the case in the Aspect Sandbox implementation – and a short examination showed that many functions would have to be modified to provide said context.

For this reason, I settled with the rather unsatisfying solution to simulate nested classes by converting them into top-level classes with fully qualified class names.

Another, more serious “problem” with the base language is that it does not support reference semantics. Examples like the one presented in [2], Fig. 3–5, are therefore not feasible (for instance, calls to `addObserver()` would never be able to modify the object on which it is called, but only a local copy). Therefore, examples that would underline the usefulness of ACI cannot be implemented in the Aspect Sandbox implementation at this time. Tests show, however, that classes are correctly weaved and advice are enabled on deployment.

The lack of good examples that show the need for higher-level aspect-oriented systems like ACI is not a problem in this context. This is not an examination of ACI, but rather one of the proposed model for aspect-oriented systems.

## 7.2 Experiences with modeling and implementation

The modeling of ACI in terms of the framework has not been unambiguous. On the way to the version presented herein, I developed a number of different possibilities that also seemed to be valid. Improvements on the implementation led to more natural ways to express the system within the model. Also, where the system did not seem to be a very good fit for the framework, these uncertainties motivated changes to the implementation in specific directions.

I also was surprised about the insights into ACI that I gained while implementing it in a rather simple and scaled-down fashion. One encounters subtle but fundamental problems with the implementation of a mechanism while implementing a simple version in the Aspect Sandbox.

An example for this kind of problem one only sees during the implementation is the issue with merging the constructors I have mentioned above. Software Engineers try to invent processes that can cope with the fact that it is next to impossible to completely specify a system up front. Why should researchers be able to completely and flawlessly specify a new mechanism as complex as ACI?

For these reasons, I believe it could be beneficial to do such an implementation when designing a new aspect-oriented system.

## 7.3 Summary of the integration into the model

I believe to have shown that ACI can be integrated in a natural and straightforward way into the model. I also believe to have presented enough details of a working implementation to show that my modeling of the mechanism can be regarded as correct.

Missing features like dynamic deployment or a wrapper mechanism as in CAESAR could be added with moderate effort.

I think this result (and hopefully more results gathered in the future, see below) should help to lend validity to the modeling framework.

## 7.4 Possible future work

Some things are left to be done. The implementation as it is now performs almost no error checking and could be much improved by adding timely and accurate error messages. Since the Aspect Sandbox as a whole is more a proof-of-concept implementation than a real development platform, that does not seem to be necessary (it also lacks a type checker, although the base language does have a type concept).

Also, like the current implementation has evolved over time (and with it the modeling in terms of the framework), it seems possible that a shorter, more concise implementation (along with a still more natural expression in the framework) could be found. However, the source code of the core weaver is quite small already.

An interesting direction for further work is to integrate more different mechanisms into the framework. While I am still not fully convinced that the model captures the essence of aspect-orientation, the natural way I have been able to incorporate another system reinforces its validity. If future work shows that more mechanisms can be added in a similarly easy fashion, the model could be used to explore the space of possible aspect-oriented systems it allows.

Such a systematic exploration of possible design decisions based on the framework could in turn lead to new insights into aspects or new types of aspect-oriented mechanisms we may not think of otherwise.

## References

- [1] MASUHARA, H., AND KICZALES, G. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP2003* (2003), pp. 2–28.
- [2] MEZINI, M., AND OSTERMANN, K. Conquering aspects with caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development* (2003), ACM Press, pp. 90–99.