Parallel Composition of Aspect Mechanisms: Design and Evaluation[*]

[Extended Abstract]

David H. Lorenz Sergei Kojarski
Department of Computer Science, University of Virginia
Charlottesville, Virginia 22904-4740, USA
{lorenz,kojarski}@cs.virginia.edu

ABSTRACT

There is a growing interest in the composition of aspect mechanisms. All extant works, however, avoid an important question: what should the semantics of the composed multi-extension language be? The problem is that the semantics for the composition is nowhere specified. Therefore, even if third-party composition of aspect mechanisms were successful, it is difficult to evaluate the correctness of the composition. In this paper, we propose the use of ASPECTJ 5 as a benchmark for evaluating and comparing composition techniques for integrating pointcut-and-advice mechanisms. If an aspect mechanism X implements the semantics of ASPECTJ and another mechanism Y implements the semantics of ASPECTWERKZ, then the semantics of the composition X+Y can be checked against the semantics of ASPECTJ 5. We present a novel parallel composition technique, and illustrate that it passes the ASPECTJ 5 benchmark.

1. INTRODUCTION

The availability of a variety of aspect mechanisms changes the design space for AOSD. Each aspect mechanism supports some unique crosscutting capability; and no single mechanism supports all cases of crosscutting. Yet notwithstanding, an aspect-oriented design solution can be tailored to a complex crosscutting concern by combining different aspect mechanisms to form new AOP functionality.

Unfortunately, little support is provided today for the integration of distinct aspect mechanisms. Even if ad hoc integration of aspect mechanisms were possible, there is no criteria by which to evaluate a composition technique. When it is impossible to validate the composition semantics, it is impossible to confidently integrate multiple aspect mechanisms to produce a desired effect, and the full potential of AOSD is not realized.

What we lack and need are better composition techniques and better methods for evaluating them. To this end, in this position paper we present a novel parallel composition technique and propose two evaluation criteria that may help compare composition techniques in terms of quality and correctness:

- *The Pluggable AOP benchmark:* The degree to which two distinct independently developed aspect mechanism X and Y can be subject to third-party composition in producing X+Y [5, 3]. This introduces a spectrum ranging from black-box to white-box composition techniques.
- *The* ASPECTJ *5 benchmark:* Demonstrate X and Y such that X implements the semantics of ASPECTJ 1.2, Y implements the semantics of ASPECTWERKZ, and the semantics of the composition X+Y yields the semantics of ASPECTJ 5.

In this position paper we propose a high-level semantical framework that passes both benchmarks.

2. REQUIREMENTS

In ASPECTJ 5, there is no conceptual difference between ASPECTJ's *code-style* and ASPECTWERKZ's *annotation-style* aspects. That is, in ASPECTJ 5, ASPECTJ and ASPECTWERKZ are just different syntactical interfaces to a single underlying mechanism. Essentially, ASPECTJ 5 roughly behaves as if it were ASPECTJ 1.2 (or ASPECTWERKZ).

A composition technique passes the ASPECTJ 5 benchmark if it yields a composition that behaves just like a single mechanism. We list four requirements that characterizes more precisely what it means to "behave as a single mechanism:"

- *Exposure of aspectual effect* [3]: the aspectual effect of each aspect mechanism should be exposed to all the other composed mechanisms. Specifically, in the composition of ASPECTJ and ASPECTWERKZ, ASPECTJ advice should be visible to, and be advised by, ASPECTWERKZ advice, and vice versa.
- *Hiding of the mechanism implementation* [3]: the internal operation of an aspect mechanism should be hidden from all other composed mechanisms. Specifically, in the composition of ASPECTJ and ASPECTWERKZ, ASPECTWERKZ aspects may not advise internal operations of the ASPECTJ mechanism (e.g., join-point object instantiation, pointcut matching, advice selection), and vice versa.

[*]This work is supported in part by NSF's *Science of Design* program under Grants Number CCF-0438971 and CCF-0609612.

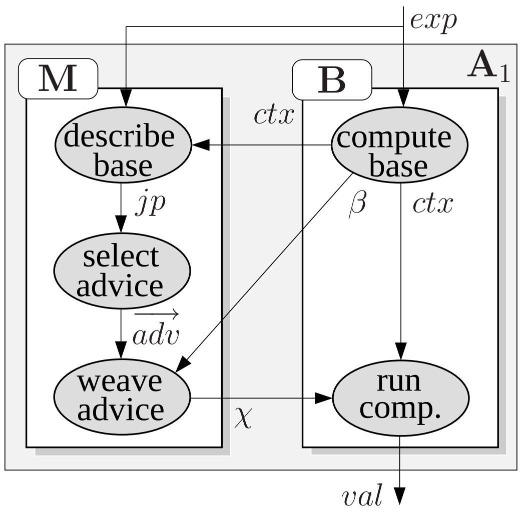


Figure 1: PA mechanism

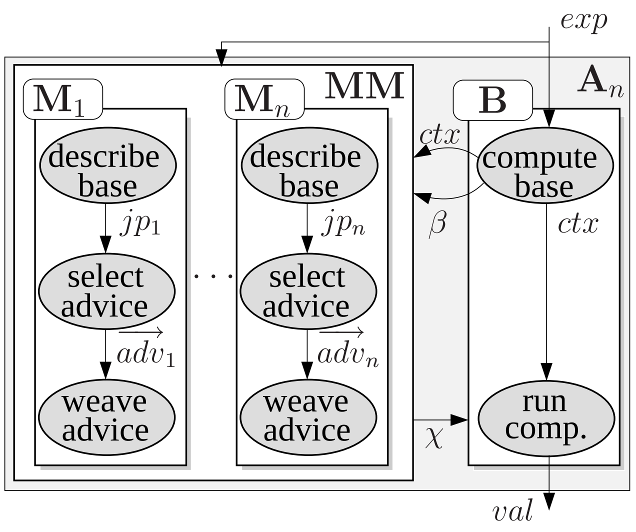


Figure 2: Multi-mechanism

- *Universal advice ordering*: pieces of advice selected by different mechanisms for the same join point should be ordered w.r.t. their types. At each join point, the composition of AspectJ and AspectWerkz first applies AspectJ and AspectWerkz pieces of **before** advice, then pieces of **around** advice, and finally pieces of **after** advice. Specifically, it is illegal for pieces of **before** advice written in one extension to be executed after **around** advice written in other extension.

- *Universal join-point history*: all the composed mechanisms should share the same program history. In the composition of AspectJ and AspectWerkz, both mechanisms observe the program execution through exactly the same sequences of join points.

3. BACKGROUND

Pluggable AOP [2, 3] is a semantical model of an AOP language in which the aspect extension semantics is defined separately from semantics of the base language. Figure 1 depicts an interpreter $\mathbf{A}_1$ that realizes the evaluation semantics for a *single-extension* AOP language $\mathcal{L}_1 = Base \times Ext$. $\mathbf{A}_1$ comprises a base mechanism $\mathbf{B}$ and an aspect mechanism $\mathbf{M}$. $\mathbf{B}$ realizes the expression evaluation semantics for the base language $Base$. $\mathbf{M}$ realizes the semantics for the PA extension Ext.

The model defines the base mechanism $\mathbf{B}$ as a computation *constructor*, and the mechanism $\mathbf{M}$ as a computation *transformer*. $\mathbf{B}$ constructs a base computation β by interpreting an input expression exp. $\mathbf{M}$ selects advice and *weaves* them by transforming β into χ, which replaces β in the program execution.

In this paper we focus on pointcut and advice (PA) aspect extensions. $\mathbf{M}$ associates base computations with advice using *join points*. A join point jp is a *description* of a base computation that allows the mechanism to identify β. More specifically, the join point is an abstraction of β's context ctx, and a currently evaluated expression exp. In AspectJ, for example, join points describe computations using dynamic (argument values, **this** and **target** objects), lexical (class and method where expression being evaluated is located), and static data (signature types) that constitutes the context of a call, an execution, and other Java computations.

At each join point the mechanism selects a set of advice pieces $\overrightarrow{adv}$. Depending on the Ext syntax, an advice $adv \in \overrightarrow{adv}$ may be a $Base$ expression, or may be written in extension-specific terms. For example, in AspectJ, advice body expressions are written in Java. COOL, on the other hand, can specify advice using only domain-specific terms (`mutex`, `selfex`, etc.)

The advice weaver in $\mathbf{M}$ produces a weaving result computation χ by transforming β. Intuitively, χ wraps advice effects around the base computation. χ proceeds to β when the advice proceeds to the base program.

3.1 Mechanisms Composition

A multi-extension AOP language $\mathcal{L}_n$ integrates $Base$ with a set of different PA extensions $Ext_1, \ldots, Ext_n$. Figure 2 presents a general approach to third-party composition of PA extensions. In Figure 2, the $\mathcal{L}_n$ semantics is realized by a multi-extension AOP interpreter $\mathbf{A}_n$. $\mathbf{A}_n$ comprises two parts: the base mechanism $\mathbf{B}$ and a *multi-mechanism* $\mathbf{MM}$. Interaction between the two is similar to the interaction between base and aspect mechanisms in the single-extension interpreter.

The multi-mechanism composes a set $\mathbf{M}_1, \ldots, \mathbf{M}_n$ of PA mechanisms that realize semantics of the $Ext_1, \ldots, Ext_n$ extensions, respectively. The mechanisms collaboratively advise base computations. A single base computation β is generally advised by a subset of composed aspect mechanisms.

The mechanisms preserve their individual features (e.g., join point ontology, pointcut matching, advice ordering) in the composition. Each of the mechanisms reflects β by creating its own join point, and selects advice pieces written in its respective aspect extension. $\mathbf{MM}$ constructs χ by integrating advice pieces selected by different mechanisms with β. The result computation χ encapsulates mechanism collaboration behavior.

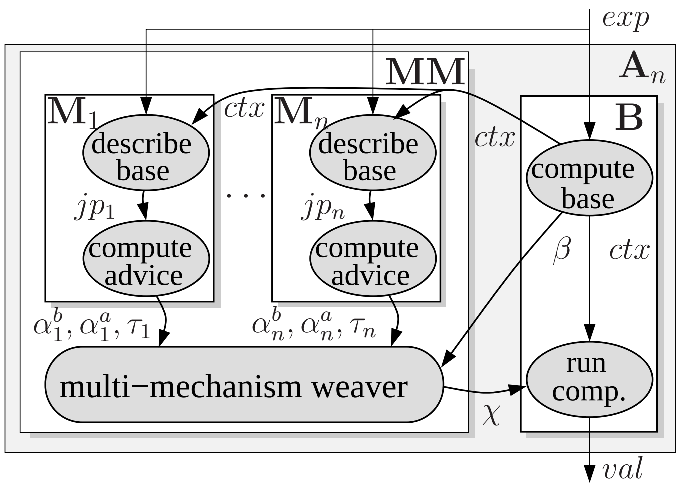


Figure 3: Parallel Composition of PA mechanisms

4. COMPOSITION FRAMEWORK

The framework specializes the general composition approach to allow ASPECTJ 5-like compositions. The framework composes PA mechanisms in *parallel*. For each base computation β, $\mathbf{MM}$ passes the currently evaluated expression exp and a computation context ctx to all the PA mechanisms $\mathbf{M}_1, \ldots, \mathbf{M}_n$. Each PA mechanism $\mathbf{M}_i$ then constructs a join point, computes an aspectual effect, and passes it to the multi-mechanism weaver. The weaver combines the aspectual effects of all the mechanisms, and wraps them around the base computation β.

The framework imposes design requirements on the PA mechanisms for supporting desired collaboration behavior. Specifically, the PA mechanism $\mathbf{M}_i$ must provide its aspectual effect via three functions, namely a before computation α_i^b, an after computation α_i^a, and an around computation transformer τ_i. Intuitively, α_i^b, τ_i, and α_i^a provide meaning for **before**, **around**, and **after** advice pieces that were selected by the mechanism, respectively.

The multi-mechanism weaver composes all constructed effects together into the result computation χ. The weaver semantics is illustrated in Figure 4. χ is built by sequencing three multi-effect computations, namely a before computation α_χ^b, an around computation α_χ^{ar}, and an after computation α_χ^a:

$$\chi = \alpha_\chi^b \triangleright \alpha_\chi^{ar} \triangleright \alpha_\chi^a$$

where $\triangleright$ denotes a left-to-right execution order for the sequenced computations: χ first executes α_χ^b, then α_χ^{ar}, and finally α_χ^a. More specifically, let γ be a composite computation defined as:

$$\gamma = \gamma_1 \triangleright \ldots \triangleright \gamma_m$$

Then γ execution runs the subcomputations $\gamma_1, \ldots, \gamma_m$ one by one, starting from γ_1 through γ_m. All subcomputations are passed the same context as that passed to γ. The value computed by γ is the value returned from γ_m.

α_χ^b sequences the before computations that are produced by the aspect mechanisms in the index-ascending order:

$$\alpha_\chi^b = \alpha_1^b \triangleright \ldots \triangleright \alpha_n^b$$

α_χ^a sequences the after computations in the index-descending order:

$$\alpha^a = \alpha_n^a \triangleright \ldots \triangleright \alpha_1^a$$

The around transformers $\tau_1, \ldots, \tau_n$ are composed sequentially, in the following index-descending order. First, τ_n produces the around computation α_n^{ar} by transforming the base computation β. The τ_{n-1} then produces α_{n-1}^{ar} by transforming α_n^{ar}. The process repeats until τ_1 produces α_1^{ar} by transforming the output of τ_2:

$$\alpha_\chi^{ar} = \alpha_1^{ar} = \tau_1(\tau_2(\ldots(\tau_n(\beta))\ldots))$$

The composition of the around transformers allows aspects written in one extension proceed to the aspects written in other extension. The α_1^{ar} proceeds to α_2^{ar}, α_2^{ar} proceeds to α_3^{ar}, and so on. β is executed only if all the around computations proceed.

5. EVALUATION

Our composition framework passes both the Pluggable AOP and the ASPECTJ 5 benchmarks. The framework is tailored for third-party extensibility.

The explicit separation between a mechanism's implementation (e.g., $\mathbf{M}_i$ internals) and its aspectual effect (e.g., $\alpha_i^b, \tau_i, \alpha_i^a$) satisfies the *hiding of the mechanism implementation* requirement. The *exposure of the aspectual effect* requirement is satisfied if the composed mechanisms evaluate advice expressions in $\mathbf{A}_n$. For example, $\mathbf{M}_i$ exposes the **before** advice expression exp_{adv}^i by constructing α_i^b as:

$$\alpha_i^b(ctx) = \mathbf{A}_n(exp_{adv}^i)$$

The execution of the α_i^b computation is the execution of the exp_{adv}^i advice expression in $\mathbf{A}_n$. Hence, the execution of exp_{adv}^i is *exposed* to the composed mechanisms.

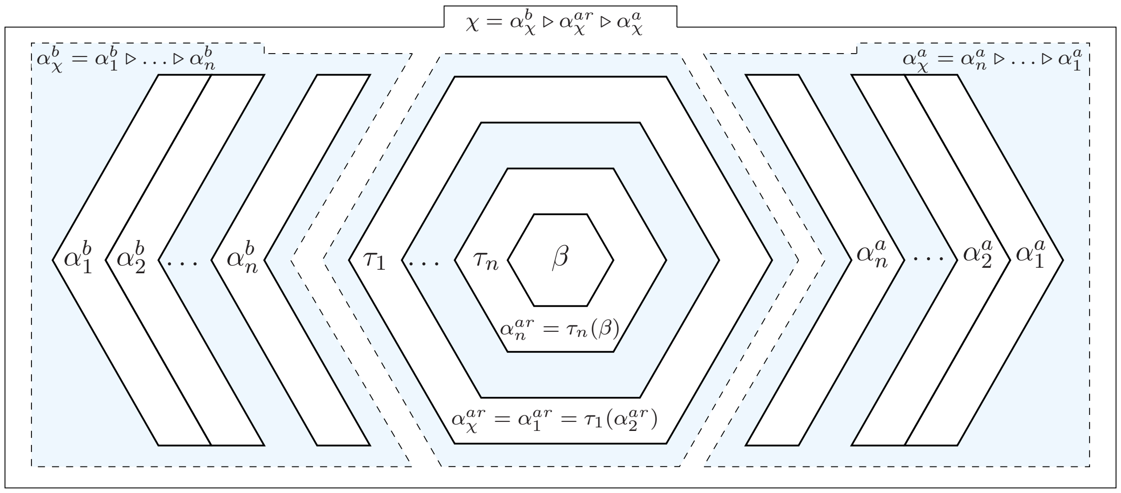
Semantics of the framework's multi-mechanism weaver component respects the *universal advice ordering* requirement. The weaver semantics enable the desired advice ordering in the composition of ASPECTJ and ASPECTWERKZ. Finally, the parallel mechanism composition architecture satisfies the *universal join-point history* requirement. All the composed mechanisms observe exactly the same sequences of computation contexts and expressions.

As a proof of concept, we implemented and composed two aspect extensions to TinyJ, a simple Java-like object-oriented language, namely, TinyAJ and TinyAW. TinyAJ is an ASPECTJ-like extension that provides its own syntax for specifying aspects. TinyAW is an ASPECTWERKZ-like extension that allows to define aspects using comments to TinyJ classes and methods. The third-party composition of TinyAJ and TinyAW satisfies all four requirements.

6. RELATED WORK

Pluggable AOP [3] is a third-party composition framework that supports the composition of arbitrary dynamic aspect mechanisms into an AOP interpreter. The framework employs a *sequential* composition of aspect mechanisms. The framework mechanisms collaborate by hiding, delegating, and exposing expression evaluation to the other mechanisms.

The sequential framework passes the Pluggable AOP composition benchmark but not the ASPECTJ 5 benchmark. Specifically, a se-



$\chi = \alpha^b_\chi \triangleright \alpha^{ar}_\chi \triangleright \alpha^a_\chi$
$\alpha^b_\chi = \alpha^b_1 \triangleright \ldots \triangleright \alpha^b_n$
$\alpha^a_\chi = \alpha^a_n \triangleright \ldots \triangleright \alpha^a_1$
α^b_1 α^b_2 $\ldots$ α^b_n
τ_1 $\ldots$ τ_n β
α^a_n $\ldots$ α^a_2 α^a_1
$\alpha^{ar}_n = \tau_n(\beta)$
$\alpha^{ar}_\chi = \alpha^{ar}_1 = \tau_1(\alpha^{ar}_2)$

Figure 4: Semantics of the multi-mechanism weaver

quential composition of PA mechanisms fails to satisfy the universal advice ordering and the universal join-point history requirements.

Tanter and Noyé [1] introduce a Reflex framework that allows one to compose PA extensions. Their work fundamentally differs from ours in the extension composition approach. Our framework constructs a multi-extension AOP interpreter by composing different aspect mechanisms. Intuitively, the interpreter weaves multi-extension aspects using *multiple* mechanism weavers with *different* semantics. The Reflex framework composes aspect extensions using a translation approach. The framework translates programs written in different aspect extensions to an *assembler* aspect extension. All translated aspects are then woven by a *single* weaver under *the same* semantics.

XAspects is another translation-based framework. Shonle et al. [4] present a framework for aspect compilation that allows to combine multiple domain-specific aspect extensions. The framework's composition semantics is to reduce all extensions to a single general-purpose aspect extension (AspectJ). Unfortunately, XAspects fails to satisfy the hiding of the mechanism implementation requirement [3].

7. CONCLUSION

In this paper, we propose the use AspectJ 5 as a data-point for evaluating and comparing composition techniques for integrating pointcut-and-advice mechanisms. We also present a novel parallel composition technique, and illustrate that it passes the AspectJ 5 benchmark. The composition yields an expression interpreter $\mathbf{A}_n$ that implements the evaluation semantics for programs in a multi-extension AOP language $\mathcal{L}_n$. The interpreter semantics passes the AspectJ 5 benchmark.

8. REFERENCES

[1] Éric Tanter and J. Noyé. A versatile kernel for multi-language aop. In *Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)*, 2005.

[2] S. Kojarski and D. Lorenz. Modeling aspect mechanisms: A top-down approach. In *Proceedings of the 28th International Conference on Software Engineering*. ICSE'06, 2006. To appear.

[3] S. Kojarski and D. H. Lorenz. Pluggable AOP: Designing aspect mechanisms for third-party composition. In R. Johnson and R. P. Gabriel, editors, *Proceedings of the 20th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 247–263, San Diego, CA, USA, Oct. 16–20 2005. OOPSLA'05, ACM Press.

[4] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain specific aspect languages. In *Companion to the 18th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 28–37, Anaheim, California, 2003. ACM Press.

[5] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002. With Dominik Gruntz and Stephan Murer.