

Modular Reasoning about Aspect-Oriented Programs

A Rely-Guarantee Approach

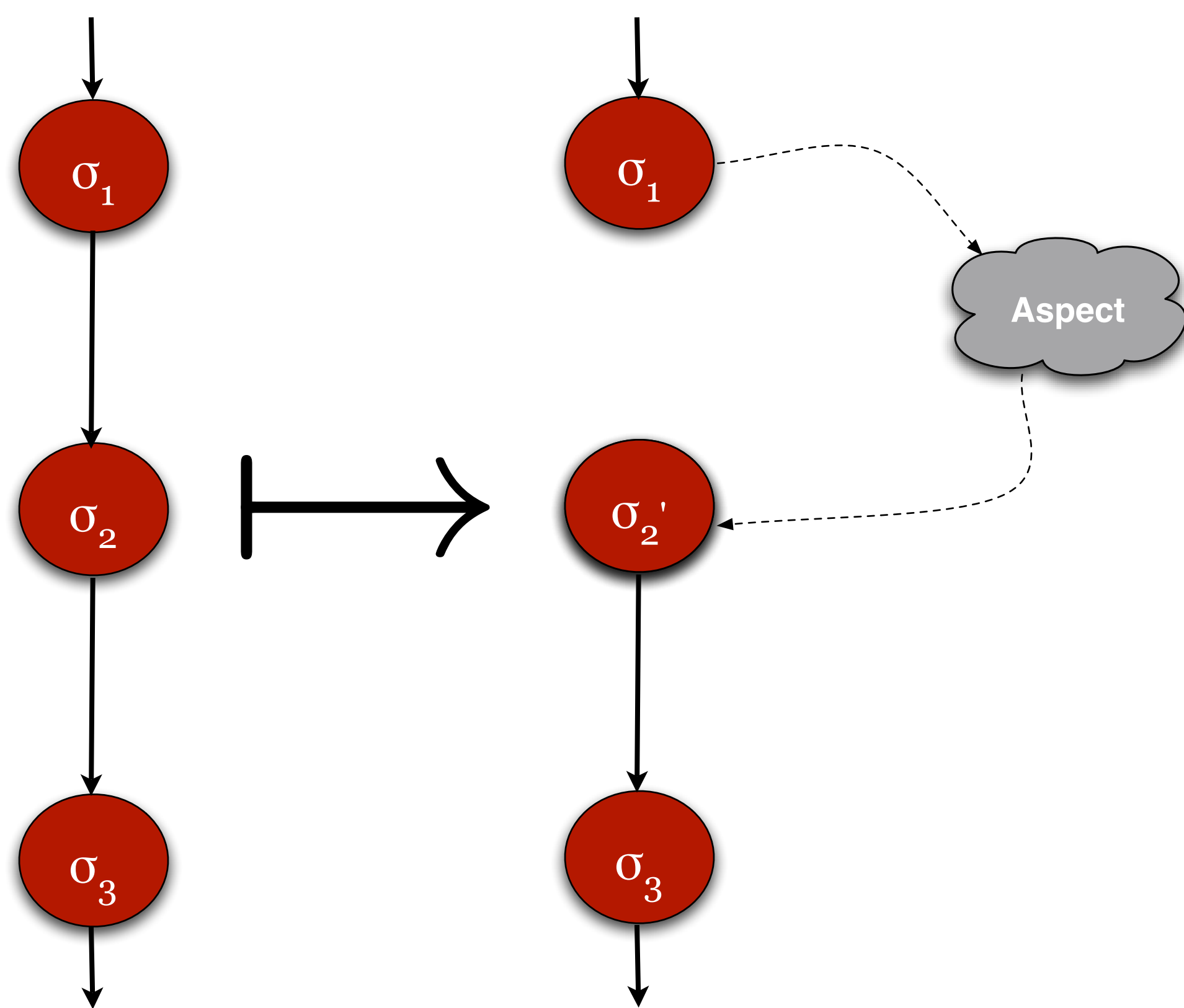
Raffi Khatchadourian and Neelam Soundarajan
The Ohio State University

Reasoning in Aspect-Oriented Programs

- AOP enables modular implementation of crosscutting concerns.
- Reasoning about AOP presents some key challenges.
- Addition of an aspect can change the behavior of the base code.
- Prior reasoning about the base code may no longer be valid.
- May be forced to reason about the entire system again accounting for the interleaving.

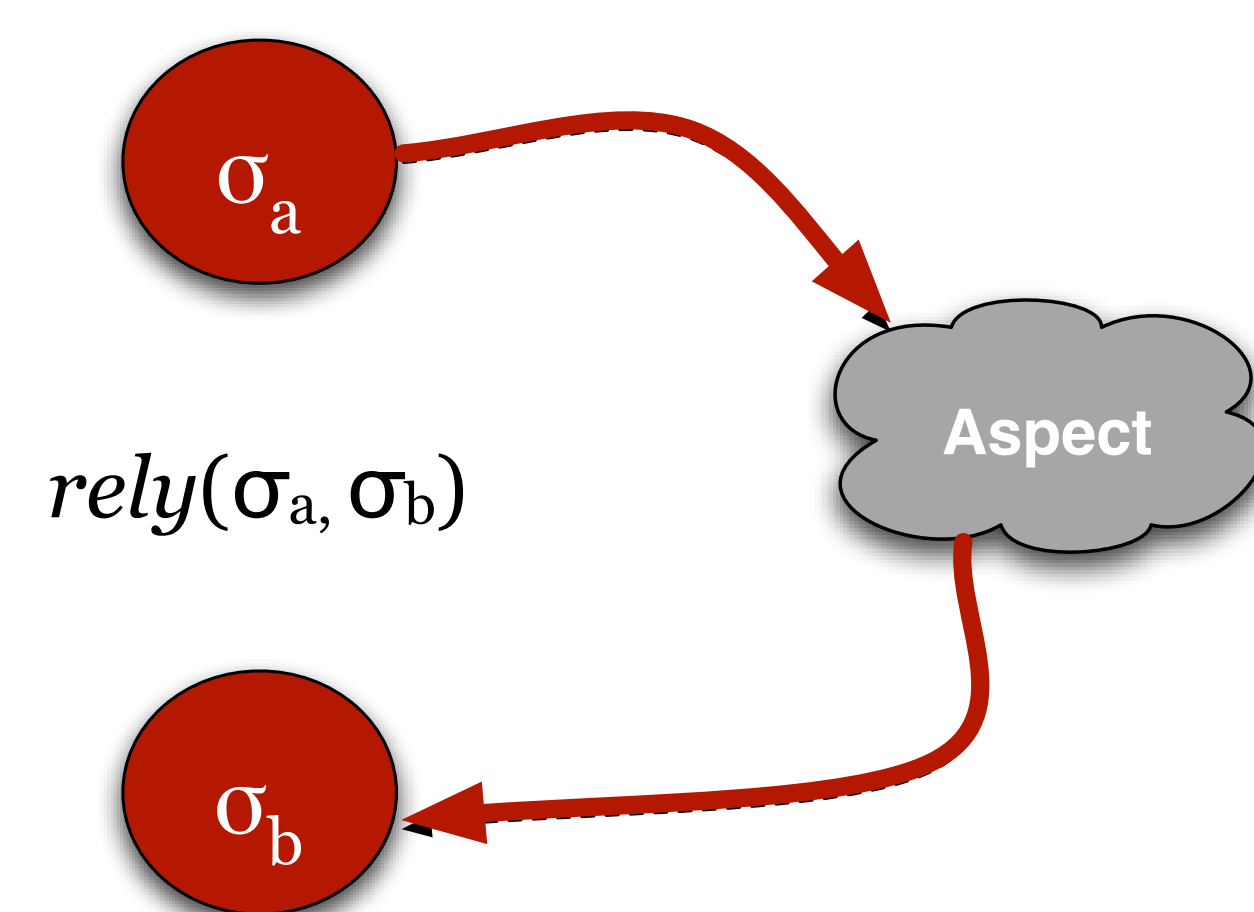
“Can we make base-code specifications more robust to aspectual changes?”

Behavior of the Classes and the Aspects



The Rely() Clause

The state at a point in the execution of a class is σ_a .



The state when the class gets control back from an aspect is σ_b .

A Rely-Guarantee Approach [Xu97] to Reasoning in AOP

A method M under the influence of advice satisfies an R/G specification denoted by

$M \text{ sat } (pre, rely, guar, post)$

if

1) M is invoked in a state which satisfies **pre** , and

2) all advice transitions satisfies **$rely$** ,

then

3) all states prior to M being intercepted by advice will satisfy **$guar$** ,

and

4) if the computation terminates, the final state will satisfy **$post$** .

A Restrictive Rely() Clause

This is “Harmless” [D&W POPL’06]

The entire state of C

$$rely(\sigma, \sigma') \equiv (\sigma = \sigma')$$

Forbids any applicable advice from making any changes in the state!

A Slightly Less Restrictive Rely() Clause

$$rely(\sigma, \sigma') \equiv (pp \Rightarrow pp^{\sigma'}_{\sigma})$$

An assertion following a joinpoint in the base-code.

Restricts the behavior of advice dependent upon the state of the base-code prior to interception.

Each occurrence of each variable of C replaced with its value in σ' rather than σ .

Case Study: A Point Zooming Aspect

```

1 class Point {
2   int x, y;
3   int s;
4
5   public Point(int xi, int yi)
6     { x=xi; y=yi; s=1; }
7   public int getX() { return (x*s); }
8   public int getY() { return (y*s); }
9
10  public void move(int nx, int ny)
11    { x=nx; y=ny; }
12 }
13
14 aspect adjustScale {
15   pointcut m(Point p):
16     execution(void Point.move(int,int))
17     && target( p );
18
19   after(Point p) : m(p) {
20     if ((p.x < 5) && (p.y < 5)) { p.s=10; }
21   }
22 }

```

Figure 1. Point Class and Aspect

Specifications

$$rely(\sigma, \sigma') \equiv [(\sigma.x = \sigma'.x) \wedge (\sigma.y = \sigma'.y)]$$

But why not: $(\sigma.s = \sigma'.s)$

- The adjustScale aspect would not meet specifications since it alters only the *scaling factor* of the Point.
- Must be sure not to impose stronger requirements than necessary on aspects that might be developed later.
- Otherwise, we may be forced to redo the task of reasoning about the class.

R/G for AOP Reasoning System

- Defines the level of “harmlessness” on a per-joinpoint basis.
- The *guar()* clause is similar to a *precondition* for advice intercepting the base-code.
- *Specification Pointcuts* (s-pointcuts) contain corresponding *rely()* and *guar()* clauses.
- Provides a way to deduce the *enriched behavior* of the aspects plus the base-code using *history variables* [Hoare78].
- Allows for modular, isolated reasoning of both the base-code and the aspects.