


Mehdi Bagherzadeh

Iowa State Univeristy

mbagherz@iastate.edu

Hridesh Rajan

Iowa State Univeristy

hridesh@iastate.edu

Gary T. Leavens

University of Central Florida

leavens@eecs.ucf.edu

Sean Mooney

Iowa State Univeristy

smooney@iastate.edu

Translucid Contracts: Expressive Specification and Modular Verification for Aspect-Oriented Interfaces

10th International Conference on Aspect-Oriented Software Development (AOSD '11), To appear

2nd Midwest Verification Day
(MVD'10)

Overview of the Talk

Modular Reasoning with Aspects in AOP is Hard.

- Any program point can be affected by aspects.
- . . . increases reasoning burden.
- Control effect of each aspect must be understood
- . . . as a result reasoning requires entire program.

Existing Work has not shown how to solve the problems.

- Do NOT sufficiently limit the scope of an aspect.
- Do NOT show how to reason about control effects.

Translucid Contracts + Quantified, Typed Events Solve It.

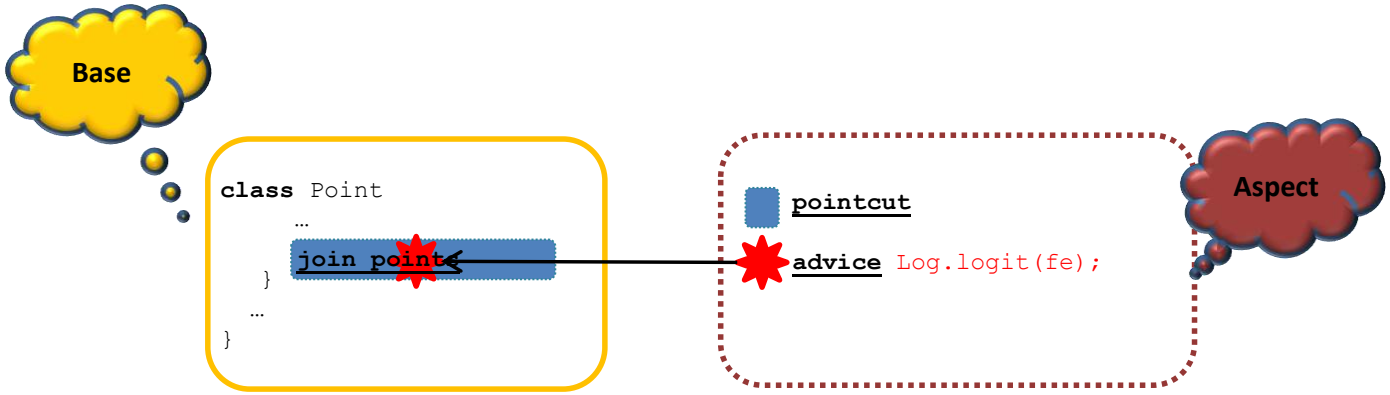
- Quantified, Typed Events narrow the scope of aspects.
- Translucid contract enable understanding of control effects.
- Together allow modular reasoning about AO programs.

AOP (Aspect Oriented Programming)

```
class Point …
  void setX(int x){
      …
      Log.logit(this);
   }
  …
}
```

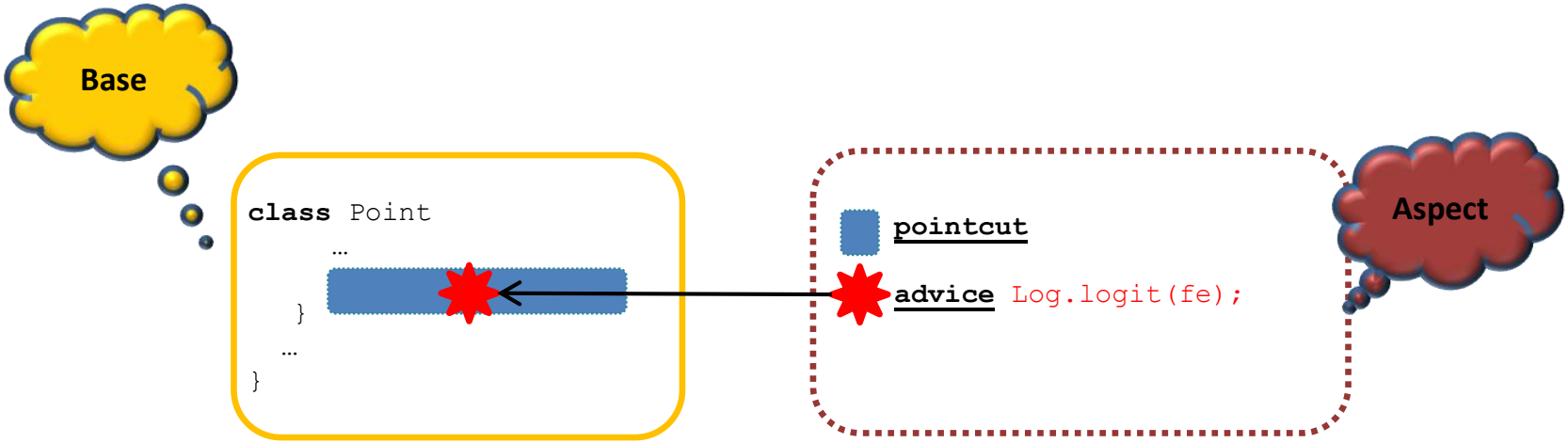
```
class Circle …
  void setR(int r){
      …
      Log.logit(this);
   }
  …
}
```

AOP (Aspect Oriented Programming)



- AO Terms: Join points, Pointcut, Advice, Aspects.

AOP (Aspect Oriented Programming)



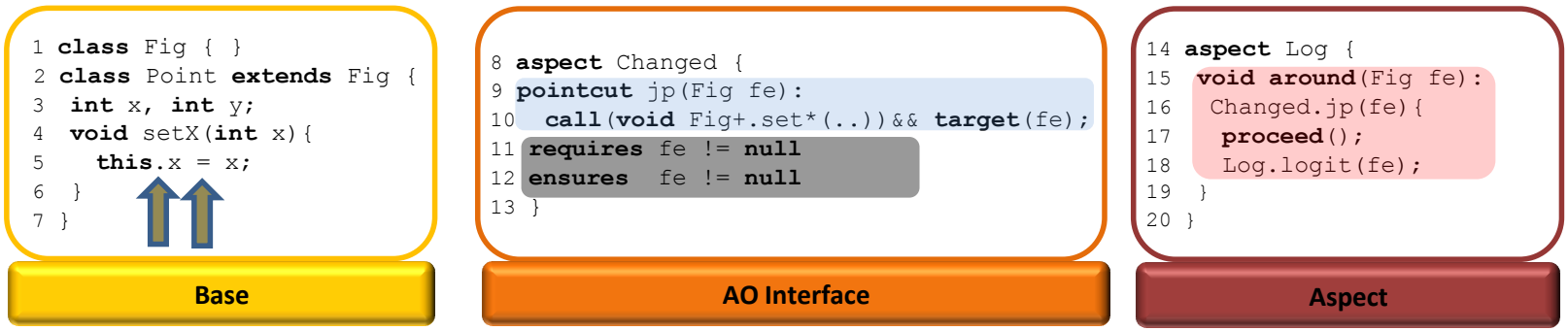
- Aspects could modify the **control flow** of the base.
- Base could be **oblivious** to the aspects.

Reasoning about AO Programs Control flow

- **All possible points** in the base that aspects can apply
- **At each point**, all the aspects that might apply there
- **For each aspect** understand its control flow

- It is a lot of work

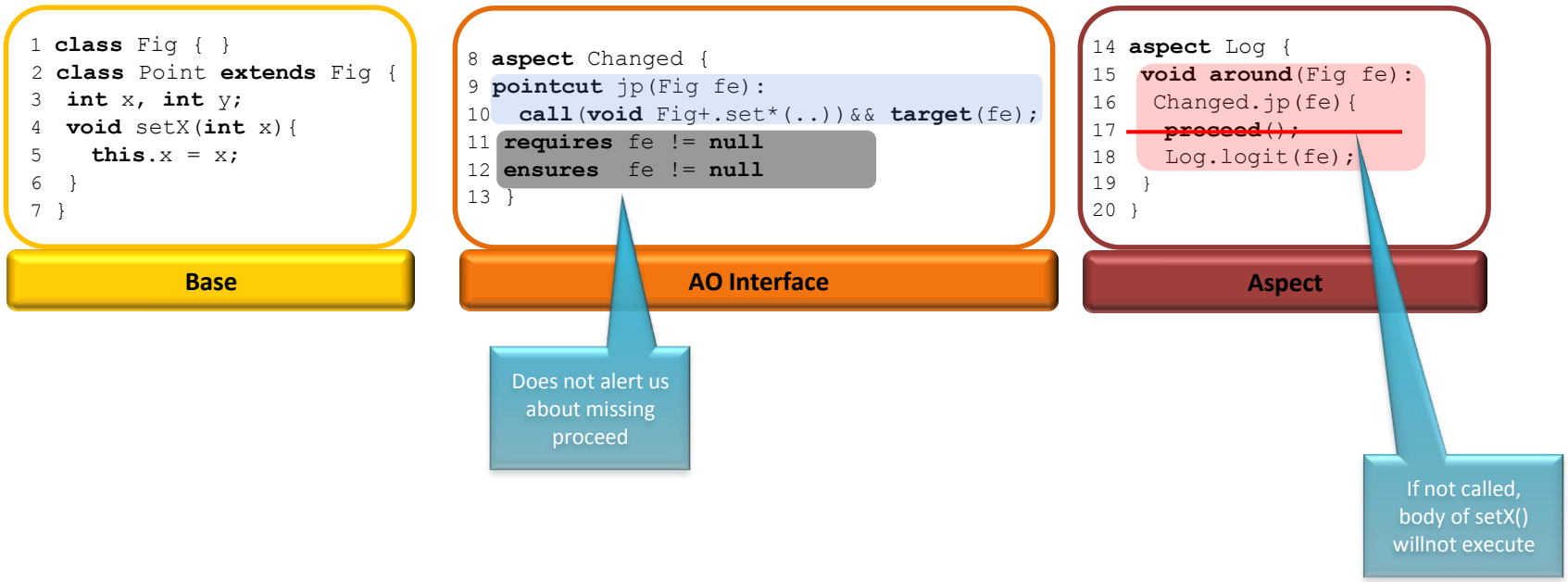
Problems: More Concretely



- **All possible points** in the base that aspects can apply
- **At each point**, all the aspects that might apply there
- **For each aspect** understand its control flow
  - Behavioral contracts fail.

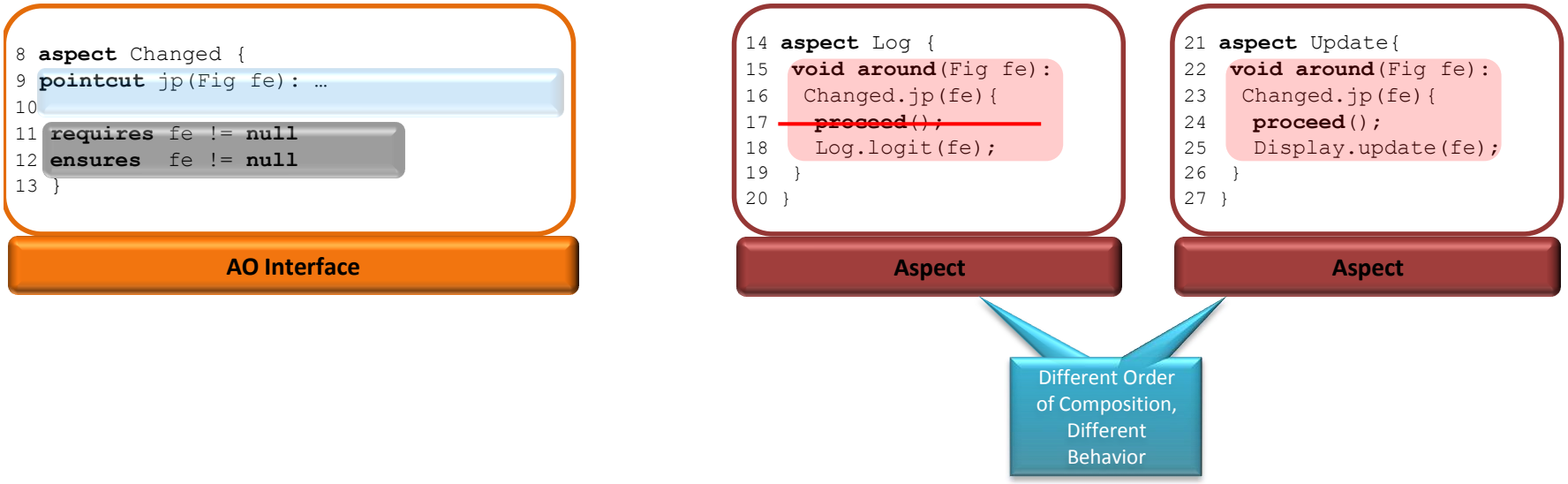
- Even this workaround has all three reasoning problems.

Control Effect Reasoning in More Detail (1)



Behavioral contracts can not talk about control effects.

Control Effect Reasoning in More Detail (2)



Behavioral contracts may not specify aspect interaction.

Ptolemy: Reconciling Reasoning and Separation.

. . . but even with all of these challenges **separation of crosscutting concerns** remains an **important** problem.

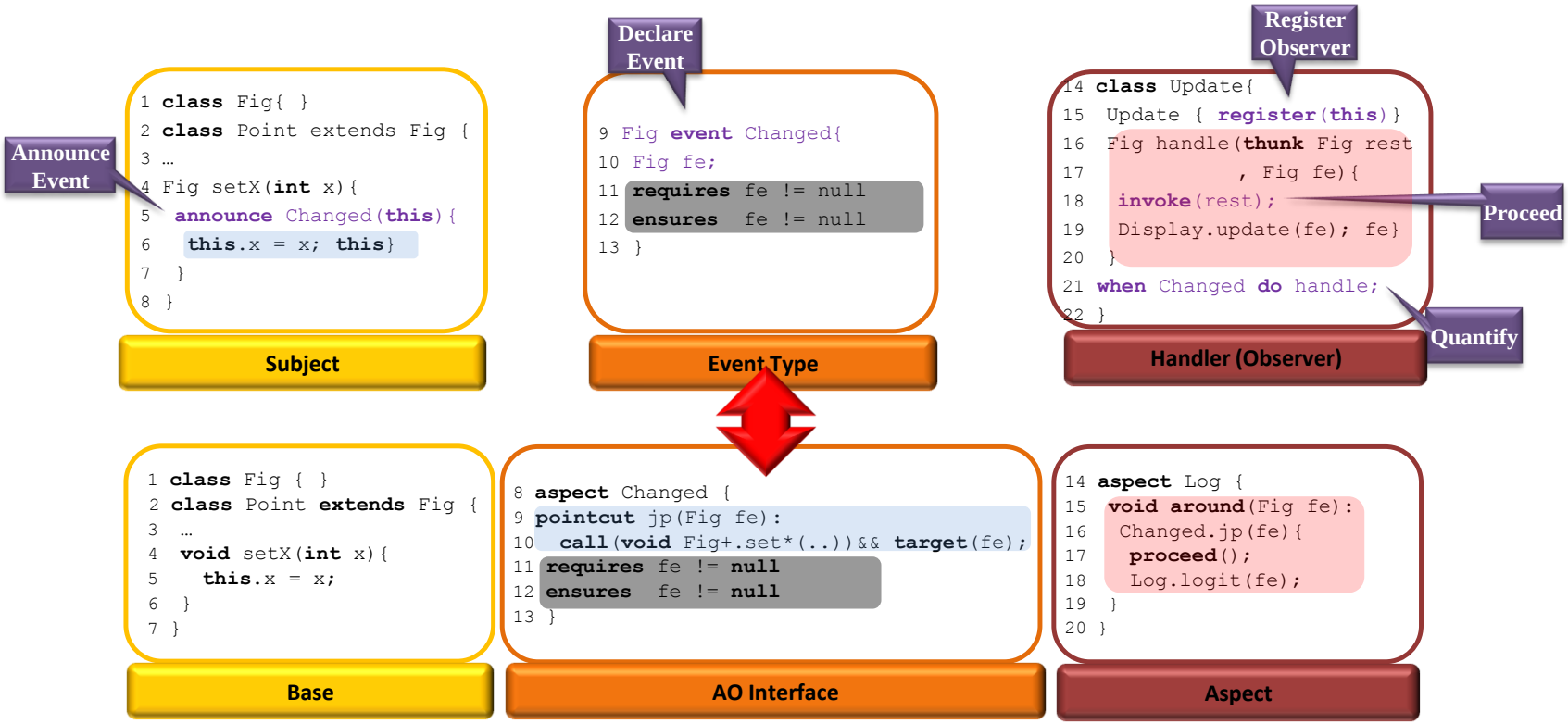
#1. Explicit, declarative, typed event announcements.

- makes module-module interaction explicit.

#2. More expressive Translucid Contracts

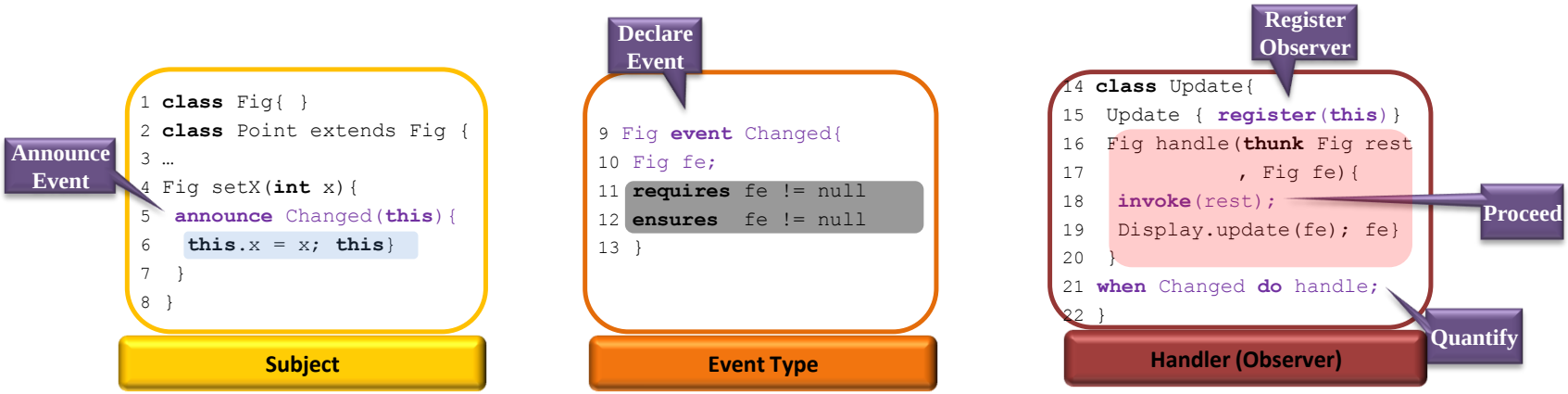
- Expose some (key) internal states.
- Facilitate modular reasoning.

Ptolemy VS AspectJ



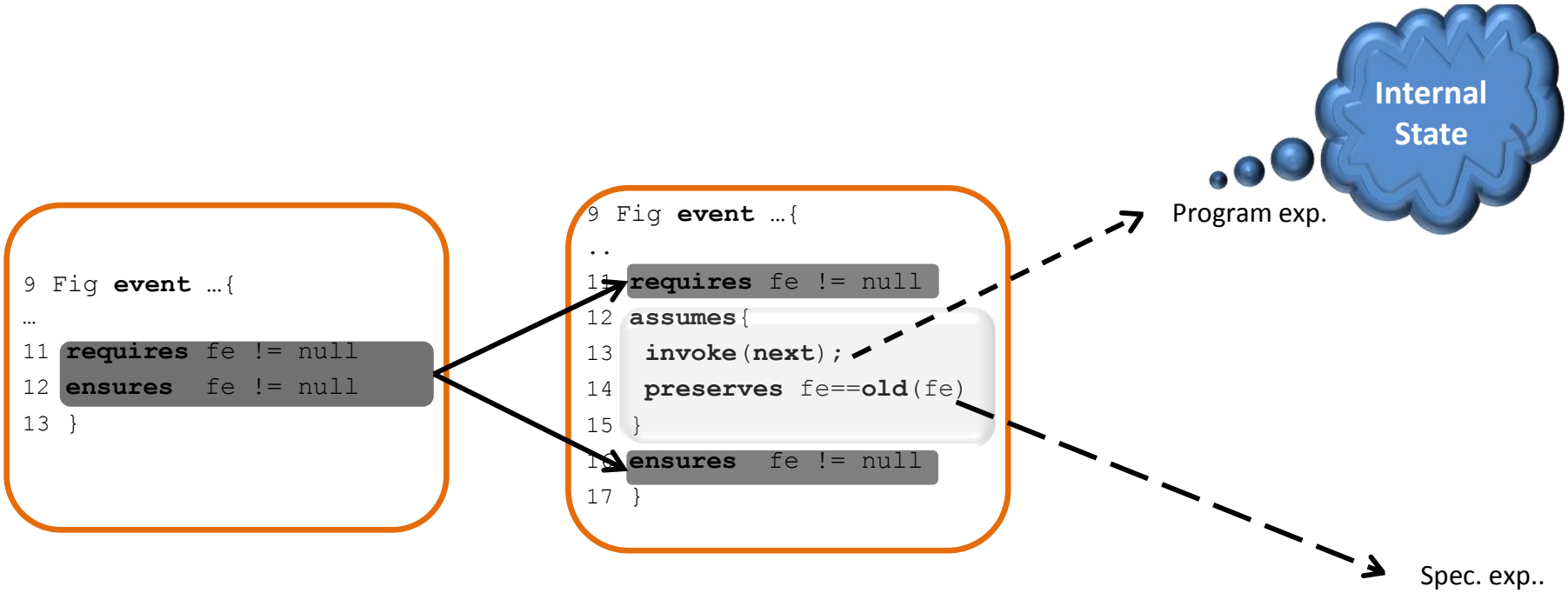
- **Subjects** announce events and **handlers** register for/handle events.

Ptolemy



- **All possible points** in the base that aspects can apply
  - Only event announcement sites
- **At each point**, all the aspects that might apply there
  - Only pre- and post-condition of translucid contract
- **For each aspect** understand its control flow
  - Translucid contracts.

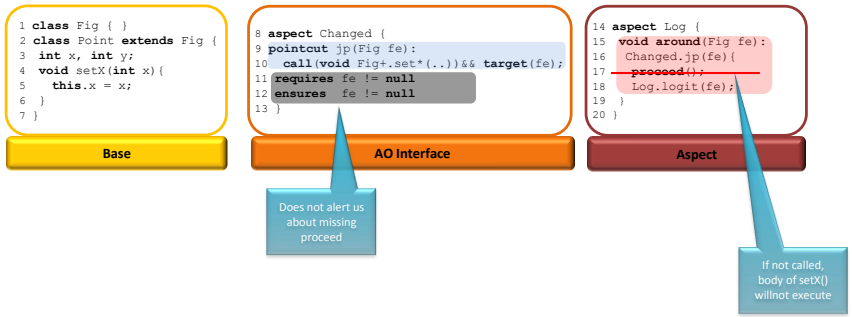
Translucid Contracts



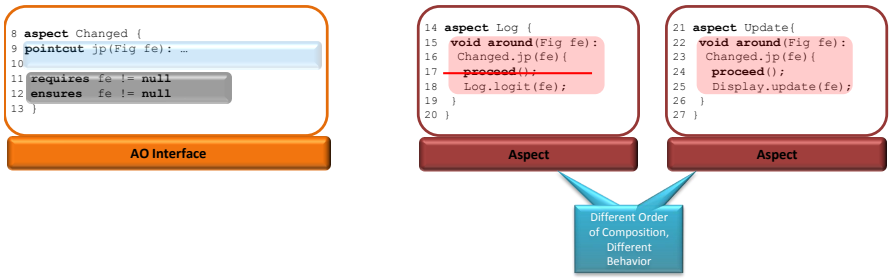
- Grey box based specification
- Assumes block is the mixture of
  - program expressions (line 13)
  - specification expressions (line 14)
- Program exp. exposes information about internal state

Revisiting the Problems with Behavioral Contracts

- **First Problem:** Behavioral contracts could not talk about control effects.
  - e.g. Behavioral contract does not alert us about missing the invoke (proceed).



- **Second Problem:** Behavioral contracts could not talk about handler's interplay.
  - Different order of composition for handlers creates different behavior.



Translucid Contracts & First Problem

```
9  Fig event Changed{
10 Fig fig;
11 requires fe != null
12 assumes{
13  invoke(next);
14  preserves fe==old(fe)
15 }
16 ensures  fe != null
17 }
```

Event

```
18 class Update{
19  …
20  Fig handle(…, Fig fe){
21   invoke(rest);
22   refining preserves fe == old(fe){
23     Display.update(fe); fe
24   }
25  }
26 when Changed do handle;
27 }
```

Aspect

- Handlers refine their translucid contract **structurally**.

Translucid contracts can specify/enforce control effects.

Translucid Contracts & the Second Problem

```
9  Fig event Changed{
10 Fig fig;
11 requires fe != null
12 assumes{
13  invoke(next);
14  preserves fe==old(fe)
15 }
16 ensures  fe != null
17 }
```

Event

```
18 class Update{
19  …
20  Fig handle(thunk Fig rest, Fig fe){
21   invoke(rest);
22   refining preserves fe == old(fe){
23     Display.update(fe); fe
24   }
25  }
26 …
27 }
```

Aspect

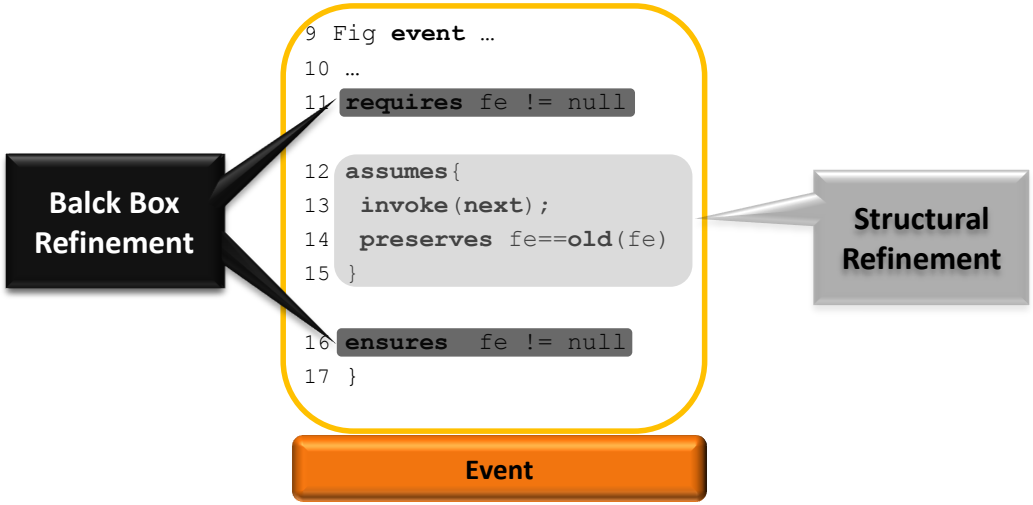
```
28 class Logging{
29  …
30  Fig handle(thunk Fig rest, Fig fe){
31   invoke(rest);
32   refining preserves fe == old(fe){
33     Log.logit(fe); fe
34   }
35  }
36 …
37 }
```

Aspect

- With invoke revealed in the contract, all of the handlers will be run, regardless of order

Translucid contracts can specify/enforce aspect's interplay.

Refinement of Translucid Contracts



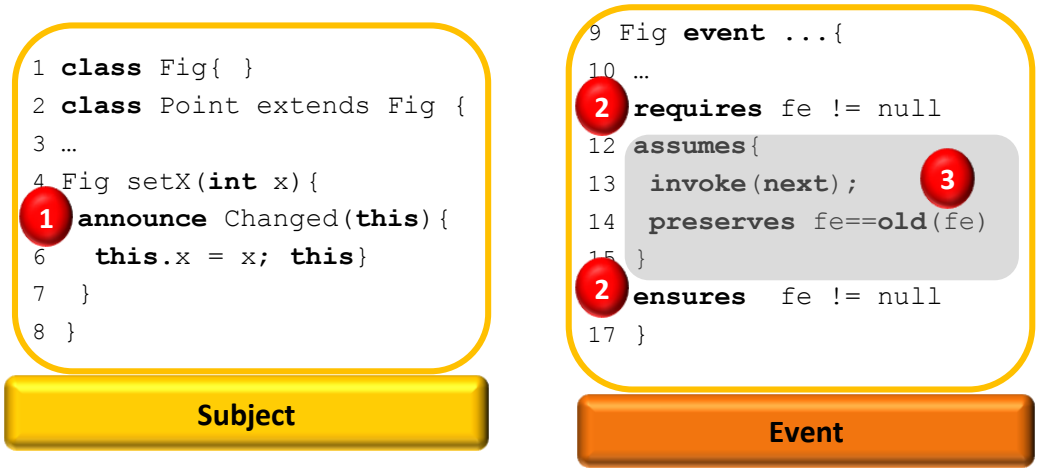
- Structural refinement
  - of the assumes block in the contract
  - by the type checker statically
- Black box refinement
  - of pre- and post-conditions of the contract
  - by runtime assertions dynamically

Refinement Requirements

	Refines Contract's
Each Handler	pre- and post-conditions
	assume block, structurally
Event Body/ Event Announcement	pre- and post-conditions

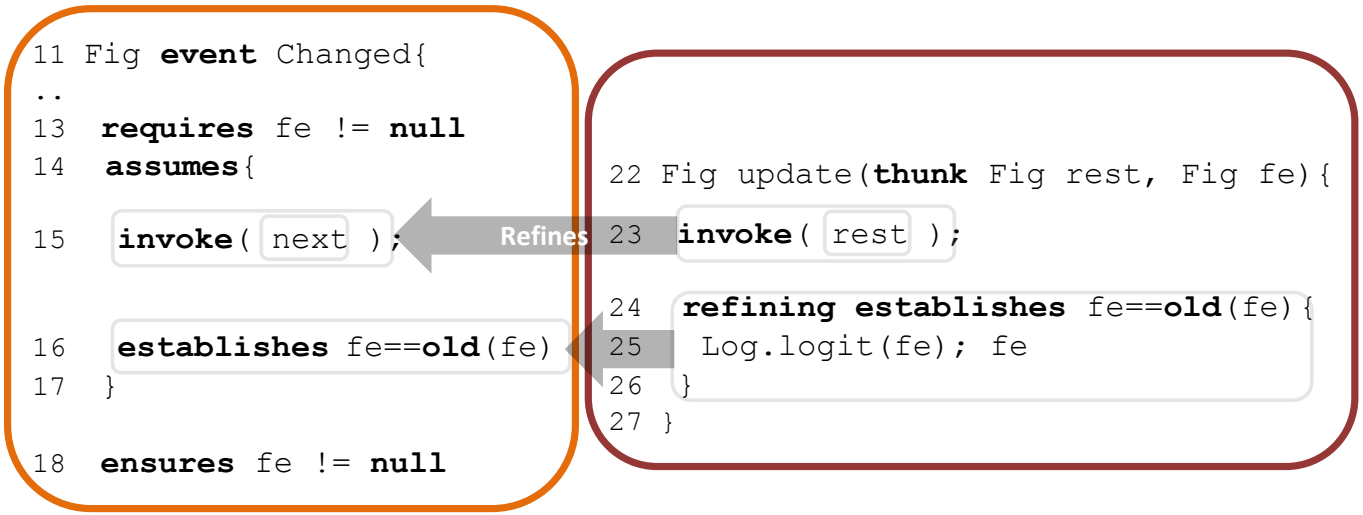
- **All handlers** refine the translucid contracts by
  - Refining contract's pre- and post-conditions
  - Structural refinement of the assumes block
- **Event body/announcement** refines translucid contract by
  - Refining contract's pre- and post-conditions
  - Reasoning about event announcement regardless of
    - number of handlers
    - handlers interplay

Reasoning about AO Programs Control flow



1. **All possible points** in the base that handler can apply
   - Explicit event announcement limits this.
2. **At each point**, all the handlers that might apply there
   - Event announcement refines pre- and post-condition of the contract.
3. **For each aspect** understand its control flow
   - Translucid contract exposes internal state to understand control effects.

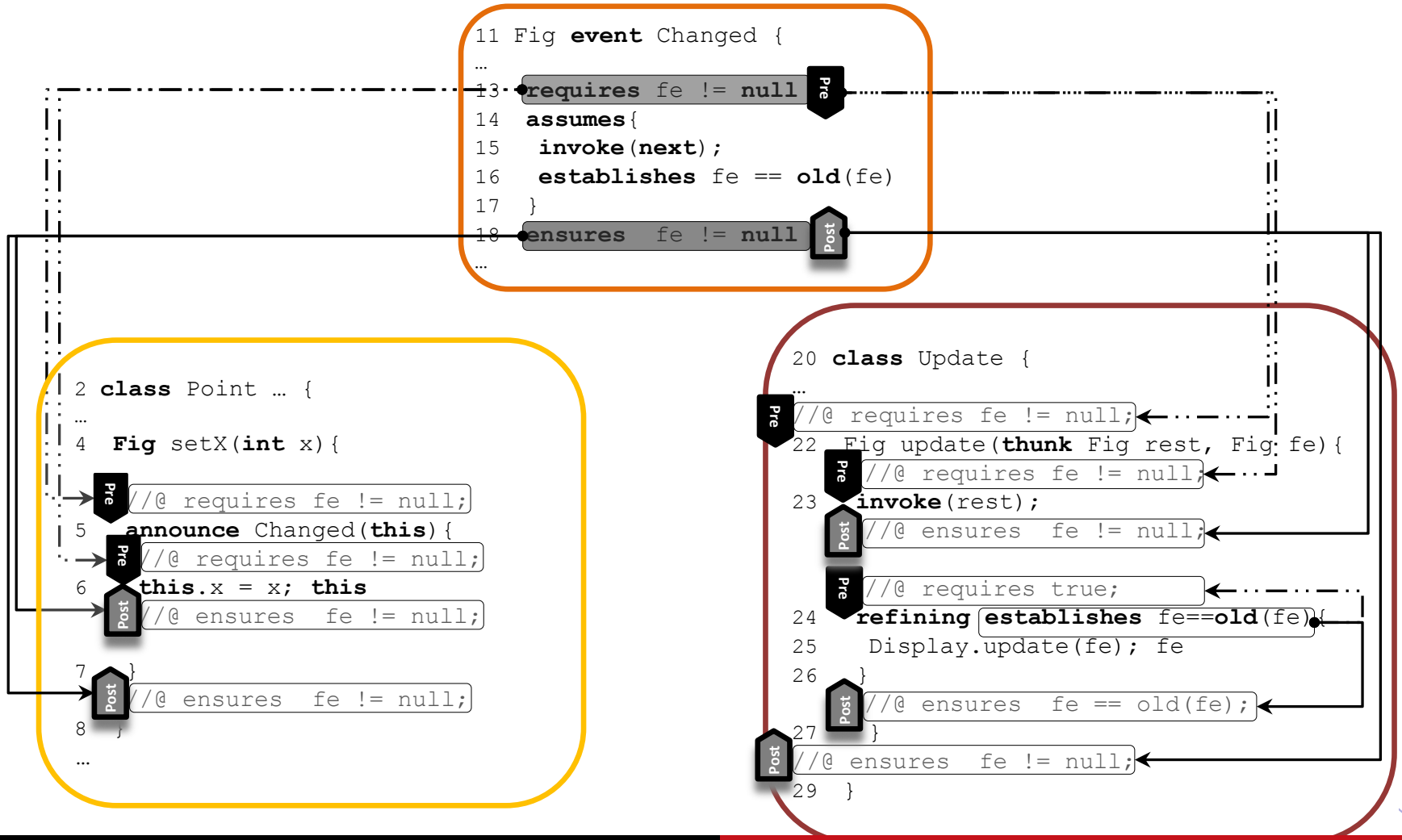
Structural Refinement of a Handler



- Textual matching of program expressions
- Black box refinement of specification expressions

Runtime Assertion Checking

Refinement of contract's pre-/post-conditions by runtime assertions



Reasoning

Translucid contracts make it possible to reason

- Subjects
- Observers
- Independent of
  - the number of the observers and
  - their order of composition

Reasoning about the Subject

```
class Point …{
 int x, int y;
 Fig setX(int x){
  announce Changed(this){
   this.x=x; this
  }
  //@ assert this.x == x
 }
}
```

```
Fig event Changed{
 Fig fig;
 requires fe != null
 assumes {
    invoke(next);
    establishes fe==old(fe)
 }
 ensures fe != null
}
```

Behavior Specification

Control Structure

- **All possible points** in the base that handlers can apply
  - Event announcement sites are the only points where handlers can apply
- **At each point**, all the handlers that might apply there
  - Translucid contracts pre- and post-condition specifies the compositional behavior of all applicable handlers
- **For each handler** understand its control flow
  - Translucid contracts specify the control effects of handlers

Reasoning about the Subject

```
class Point …{
 int x, int y;
 Fig setX(int x){
  announce Changed(this){
   this.x=x; this
  }
  //@ assert this.x == x
 }
}
```

```
Fig event Changed{
 Fig fig;
 requires fe != null
 assumes {
    invoke(next);
    establishes fe==old(fe)
 }
 ensures fe != null
}
```

Behavior Specification

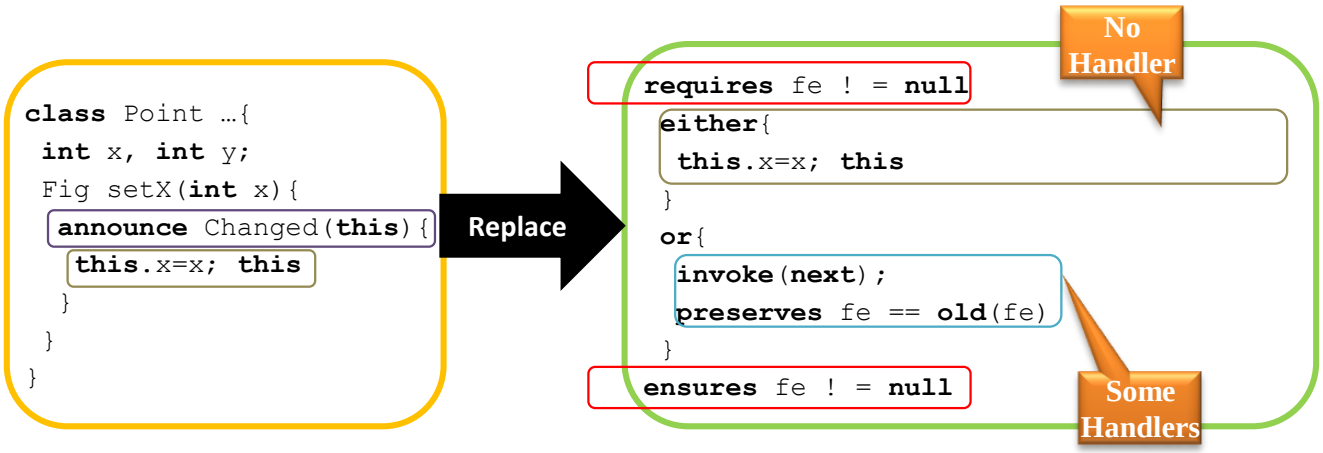
Control Structure

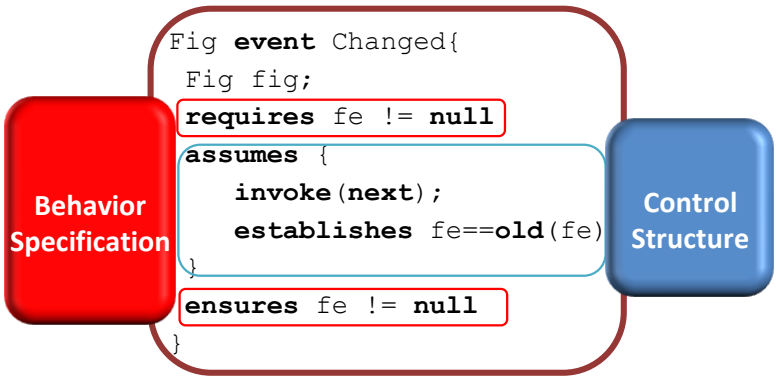
- Structure of each handler is known
- Pre- and post-condition of event announcement is known

Reasoning about the Subject

Replace every announce/invoke by specification for

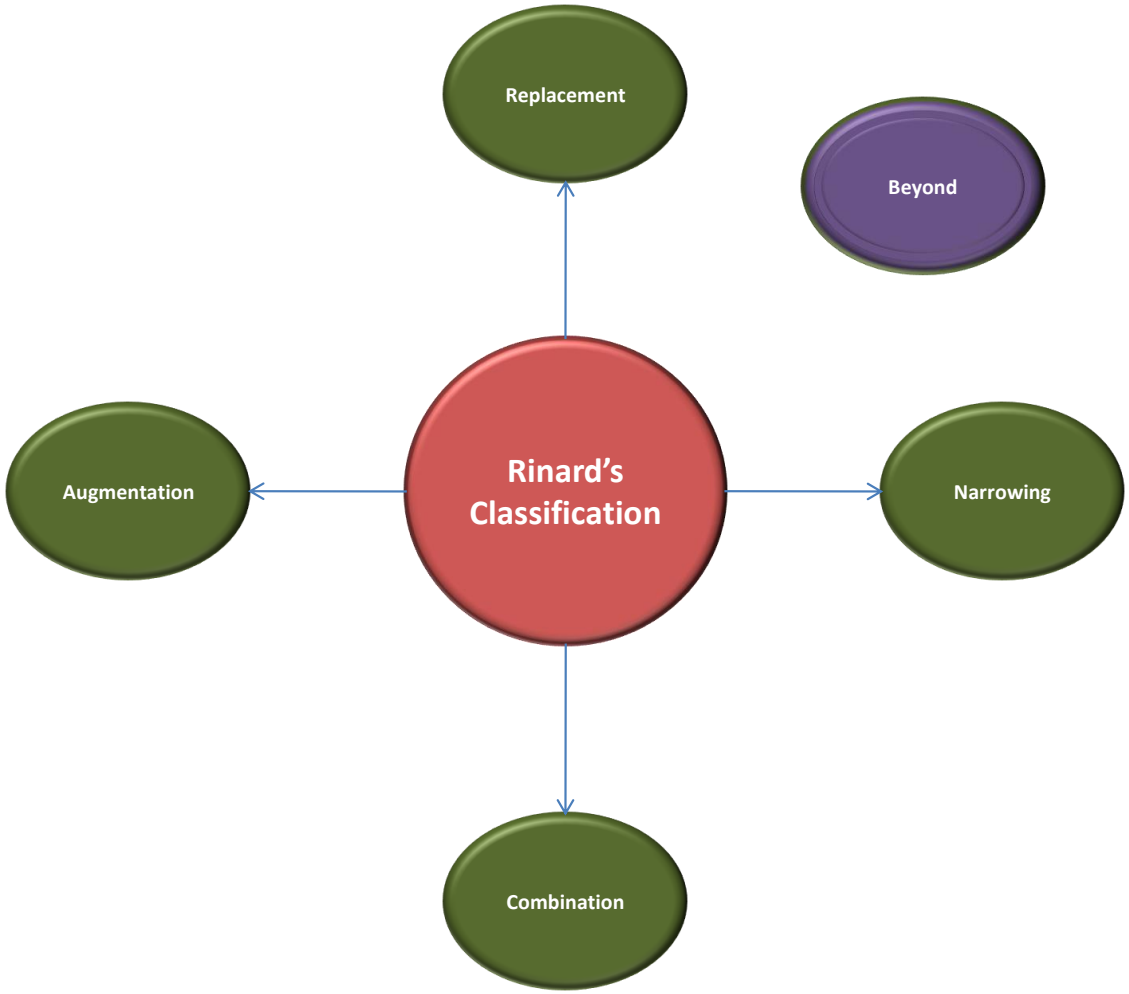
- When there is **no handler**
- When there are **some handlers**





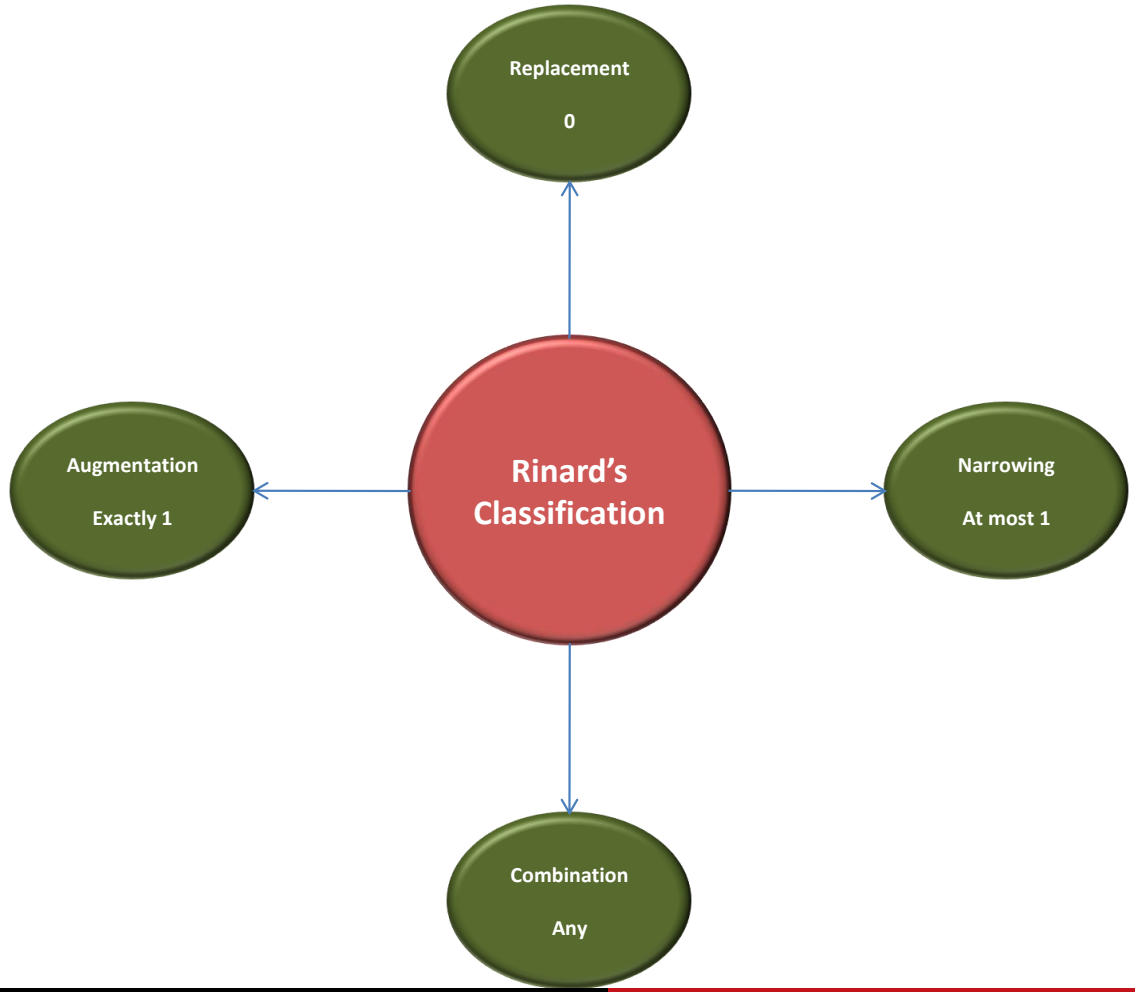
Expressiveness of Translucid Contracts

What kind of control effects, translucid contracts, can specify?



Expressiveness of Translucid Contracts

- Rinard*et al.*'s control effects classification talks about:
  - How many times **invoke** expression is evaluated in the handler.



Augmentation

Scenario: Log the changes after a figure is changed

```
1 class Fig{ }
2 class Point {
3  int x, int y;
4  Fig setX(int x){
5   announce Changed(this){
6    this.x = x; this}
7  }
8 }
```

```
9 Fig event Changed{
10 Fig fig;
11 requires fe != null
12 assumes{
13  invoke(next);
14  preserves fe==old(fe)
15 }
16 ensures  fe != null
17 }
```

```
18 class Update{
19  Update { register(this)}
20  Fig handle(thunk Fig rest, Fig fe){
21   invoke(rest);
22   refining preserves fe == old(fe){
23    Log.logit(fe); fe
24  }
25  }
26 when Changed do handle;
27 }
```

Exactly one invoke

Narrowing

Scenario: Modify the figure only if its modifiable (not fixed)

```
1 Fig event Changed{
2  Fig fe;
3  requires fe != null
4  assumes{
5   if(fe.fixed != 0)
6    invoke(next)
7   else
8    establishes fe==old(fe)
9  }
10  ensures  fe != null
11 }
```

At most one invoke

```
12 class Enforce{
13  Fig check(thunk Fig rest, Fig fe){
14   if(fe.fixed != 0)
15    invoke(rest)
16   else
17    refining establishes fe==old(fe){
18     fe
19    }
20  }
21  when Changed do check;
22 }
```

```
23 class Fig { int fixed; }
```

Beyond Rinard*et al.*'s Classification

Some scenario's which could not be specified using Rinard's classification
Scenario:

- Scaling factor $\in \{\mathbf{1}, \mathbf{10}\}$
- Point scales only if its close enough to origin

Beyond Rinard*et al.*'s Classification

```
1  Fig event Moved{
2   Point p;
3   requires p != null
4   assumes{
5    invoke(next);
6    if(p.x<5 && p.y<5)
7     establishes p.s==10
8    else
9     establishes p.s==1
10  }
11  ensures  p != null
12 }
```

```
13 class Scaling{
14  Fig scaleit(thunk Fig rest,
15              Point p){
16   invoke(rest);
17   if(p.x<5 && p.y<5)
18    refining establishes p.s==10{
19     p.s = 10; p
20    }
21   else
22    refining establishes p.s==1{
23     p.s = 1; p
24    }
25  }
26  when Moved do scaleit;
27 }
```

```
28 class Point{
29  int x, int y, int s;
30  …
31  int getX(){x*s}
32  int getY(){y*s}
33  Fig move(int x, int y){
34   announce Moved(this){
35    this.x = x; this.y = y; this
36   }
37  }
38 }
```

Applicability

Our approach is applicable to other AO interfaces

- Open Modules, AAI, etc.

Related Work

Contracts for Aspects: XPI [Sullivan*et al.*'05, '09], Cona [Skotiniotis, Lorenz '04], Pipa [Zharo, Rinard '03] and Rinard's [Rinard*et al.*'04]

- Limited behavioral contracts
- No account of aspects interplay

Modular Reasoning: EffectiveAdvice [Oliviera*et al.*'10], Explicit Joint Points [Hoffman, Eugster '07], Join Point Types [Steimann, Pawlitzki'07]

- No formally expressed and enforced contracts

Grey Box Specification and Verification: [Barnett, Schulte '01, '03], [Wasserman, Blum '97], [Tyler, Soundarajan '03]

- Our work is the first to consider grey box specification to enable modular reasoning about the code that announces/handles events

Conclusion: What I Told You

- Modular Reasoning with aspects in AOP is hard.
- Existing Work has not shown how to solve both problems.
- Translucid Contracts + Quantified, Typed Events solve it.

Modular Reasoning	All Proram Points	All Applicable Aspect	Each Aspect's Control Effects
Plain AO	Y	Y	Y
AO Interface + Behavioral Contract	Y	Y	Y
Translucid Contract + Event Types	N	N	Y

Questions

Translucid Contracts: Expressive Specification and Modular Verification for Aspect-Oriented Interfaces,

10th International Conference on Aspect-Oriented Software Development (AOSD '11), Porto de Galinhas, Brazil

Ptolemy's Syntax

$$
\begin{array}{lll}
\textit{prog} & ::= & \textit{decl}^*\ e \\
\textit{decl} & ::= & \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \textit{field}^*\textit{meth}^*\ \textit{binding}^*\ \} \\
 & & \mid t\ \texttt{event}\ p\ \{\ \textit{form}^*\ \textit{contract}\ \} \\
\textit{field} & ::= & t\ f\texttt{;} \\
\textit{meth} & ::= & t\ m\ (\textit{form}^*)\ \{\ e\ \} \mid t\ m\ (\texttt{thunk}\ t\ \textit{var}\texttt{,}\ \textit{form}^*)\ \{\ e\ \} \\
\textit{form} & ::= & t\ \textit{var}, \quad \textbf{where}\ \textit{var} \neq \texttt{this} \\
\textit{binding} & ::= & \texttt{when}\ p\ \texttt{do}\ m \\
e & ::= & n \mid \textit{var} \mid \texttt{null} \mid \texttt{new}\ c\ () \mid e\texttt{.}m\ (\ e^*\ ) \mid e\texttt{.}f \mid e\texttt{.}f = e \\
 & & \mid \texttt{if}\ (\textit{ep})\ \{\ e\ \}\ \texttt{else}\ \{\ e\ \} \mid \texttt{while}\ (\textit{ep})\ \{\ e\ \} \mid \texttt{cast}\ c\ e \\
 & & \mid \textit{form} = e\texttt{;}\ e \mid e\texttt{;}\ e \mid \texttt{register}\ (\ e\ ) \mid \texttt{invoke}\ (\ e\ ) \\
 & & \mid \texttt{announce}\ p\ (\ e^*\ )\ \{\ e\ \} \mid \texttt{refining}\ \textit{spec}\ \{\ e\ \} \\
\textit{ep} & ::= & n \mid \textit{var} \mid \textit{ep}\texttt{.}f \mid \textit{ep}\ \texttt{!=}\ \texttt{null} \mid \textit{ep} == n \mid \textit{ep} < n \mid\ \texttt{!}\ \textit{ep} \mid \textit{ep}\ \texttt{\&\&}\ \textit{ep}
\end{array}
$$

$$
\begin{array}{rcl}
\boldsymbol{n} & \in & \mathcal{N}, \text{ the set of numeric, integer literals} \\
\boldsymbol{c}, \boldsymbol{d} & \in & \mathcal{C}, \text{ a set of class names} \\
\boldsymbol{t} & \in & \mathcal{C} \cup \{\textbf{int}\}, \text{ a set of types} \\
\boldsymbol{p} & \in & \mathcal{P}, \text{ a set of event type names} \\
\boldsymbol{f} & \in & \mathcal{F}, \text{ a set of field names} \\
\boldsymbol{m} & \in & \mathcal{M}, \text{ a set of method names} \\
\boldsymbol{var} & \in & \{\texttt{this}\} \cup \mathcal{V}, \mathcal{V} \text{ is a set of variable names}
\end{array}
$$

Specification Feature

contract ::= **requires** *sp* **assumes** { *se* } **ensures** *sp*
spec ::= **requires** *sp* **ensures** *sp*
sp ::= *n* | *var* | *sp*.*f* | *sp* != **null** | *sp* == *n*
| *sp* == **old**(*sp*) | ! *sp* | *sp* && *sp*
| *sp* < *n*

se ::= *sp* | **null** | **new** *c*() | *se*.*m*(*se**) | *se*.*f* | *se*.*f* = *se*
| **if** (*sp*) { *se* } **else** { *se* } | **while** (*sp*) { *se* }
| **cast** *c se* | *form* = *se*; *se*|*se*; *se*
| **register**(*se*) | **invoke** (*se*) | **announce** *p* (*e**) { *e* }
| **next** | *spec* | **either** { *se* } **or** { *se* }

Figure: Syntax for writing translucid contracts