Mehdi Bagherzadeh
Iowa State University
mbagherz@iastate.edu

Hridesh Rajan
Iowa State University
hridesh@iastate.edu

Gary T. Leavens
University of Central Florida
leavens@eecs.ucf.edu

Sean Mooney
Iowa State University
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.

http://www.cs.iastate.edu/~ptolemy/
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.

```java
class Point {
    ... join point ...
    ...
}

Aspect Base

```
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.
Behavioral contracts cannot talk about control effects.
Behavioral contracts may not specify aspect interaction.
... 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.
Subjects announce events and handlers register for/handle events.
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.

http://www.cs.iastate.edu/~ptolemy/
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

Fig event ...

9 Fig event ...{
...
11 requires fe != null
12 ensures fe != null
13 }

9 Fig event ...{
...
11 requires fe != null
12 ensures fe != null
13 }

Program exp.
Spec. exp..
Internal State

▶ 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

http://www.cs.iastate.edu/~ptolemy/
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.
Handlers refine their translucid contract **structurally**.

Translucid contracts can specify/enforce control effects.
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

---

### Example Code

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

- **Balck Box Refinement**
- **Structural Refinement**

---

**Related Links**

- [Translucid Contracts for Aspect-oriented Interfaces](http://www.cs.iastate.edu/~ptolemy/)
## Refinement Requirements

<table>
<thead>
<tr>
<th></th>
<th>Refines Contract’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each Handler</td>
<td>pre- and post-conditions</td>
</tr>
<tr>
<td></td>
<td>assume block, structurally</td>
</tr>
<tr>
<td>Event Body/Event Announcement</td>
<td>pre- and post-conditions</td>
</tr>
</tbody>
</table>

- **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

---

http://www.cs.iastate.edu/~ptolemy/
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.

---

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

```
9 Fig event ...{
10 ...
11 requires fe != null
12 assumes{
13   invoke(next);
14   preserves fe==old(fe)
15 }
16 ensures ... extends Fig { 
3 ...
4 Fig setX(int x){ 
5   announce Changed(this){
6     this.x = x; this}
7   }
8 }
```

---

http://www.cs.iastate.edu/~ptolemy/
Textual matching of program expressions
Black box refinement of specification expressions
Refinement of contract’s pre-/post-conditions by runtime assertions

```java
11 Fig event Changed {
...  
13  requires fe != null
14  assumes{
15   invoke(next);
16   establishes fe == old(fe) 
17 } 
18  ensures fe != null
... 
```

```java
20 class Update {
... 
22 Fig update(thunk Fig rest, Fig; fe){
//@ requires fe != null 
23   invoke(rest);
//@ ensures  fe != null; 
24   refining establishes fe==old(fe) {  
25     Display.update(fe); fe 
26   }  
//@ ensures  fe == old(fe);  
27 }  
//@ ensures  fe != null; 
29 }
```

```java
2 class Point ... {
... 
4   Fig setX(int x){
//@ requires fe != null; 
5   announce Changed(this){
//@ requires fe != null; 
6   this.x = x; this 
//@ ensures fe != null; 
7 } 
//@ ensures fe != null;
8 } ... 
```

http://www.cs.iastate.edu/~ptolemy/
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

- **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

```java
class Point {
    int x, int y;
    Fig setX(int x) {
        announce Changed(this) {
            this.x=x; this
        }
        //@ assert this.x == x
    }
}
```
Reasoning about the Subject

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

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

```java
Fig event Changed{
    Fig fig;
    requires fe != null
    assumes {
        invoke(next);
        establishes fe==old(fe)
    }
    ensures fe != null
    announce Changed(this) {
        this.x = x; this
    }
    // @ assert this.x == x
}
```
Replace every announce/invoke by specification for

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

```java
class Point {
    int x, int y;
    Fig setX(int x) {
        announce Changed(this) {
            this.x = x; this
        }
    }
}
```

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

- **Behavior Specification**
- **Control Structure**

**Reasoning about the Subject**

No Handler

Some Handlers
Expressiveness of Translucid Contracts

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

Rinard’s Classification
- Replacement
- Beyond
- Augmentation
- Narrowing
- Combination
Rinard et al.'s control effects classification talks about:
- How many times `invoke` expression is evaluated in the handler.

![Diagram](http://www.cs.iastate.edu/~ptolemy/)
Scenario: Log the changes after a figure is changed

```java
9 class Fig { }
10 class Point {
11   int x, int y;
12   Fig setX(int x) {
13     this.x = x; this
14   }
15 }
16 class Update{
17   Update { register(this)}
18   Fig handle(thunk Fig rest, Fig fe){
19     invoke(rest);
20     refining preserves fe == old(fe){
21       Log.logit(fe);
22       fe
23     }
24   }
25 class Fig event Changed{
26   Fig fig;
27   requires fe != null
28   assumes{
29     invoke(next);
30     preserves fe == old(fe)
31   }
32   ensures fe != null
33 }
34 when Changed do handle;
35 }
```

Exactly one invoke
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       refining establishes fe==old(fe){
18         fe
19       }
20   }
21   ensures fe != null
22 }
```

```
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 $\in \{1, 10\}$
- Point scales only if its close enough to origin
Background Refinement and Verification Reasoning

Beyond Rinard et al.’s Classification

```java
class Point{
    int x, int y, int s;
    ...
    int getX(){x*s}
    int getY(){y*s}
    Fig move(int x, int y){
        announce Moved(this){
            this.x = x; this.y = y; this
        }
    }
}
```

```java
Fig event Moved{
    Point p;
    requires p != null
    assumes{
        invoke(next);
        if(p.x<5 && p.y<5)
            establishes p.s==10
        else
            establishes p.s==1
    }
    ensures p != null
}
```

```java
class Scaling{
    Fig scaleit(thunk Fig rest, Point p){
        invoke(rest);
        if(p.x<5 && p.y<5)
            refining establishes p.s==10{
                p.s = 10; p
            }
        else
            refining establishes p.s==1{
                p.s = 1; p
            }
    }
    when Moved do scaleit;
}
```

http://www.cs.iastate.edu/~ptolemy/
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.

<table>
<thead>
<tr>
<th>Modular Reasoning</th>
<th>All Program Points</th>
<th>All Applicable Aspect</th>
<th>Each Aspect’s Control Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain AO</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>AO Interface + Behavioral Contract</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Translucid Contract + Event Types</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

http://www.cs.iastate.edu/~ptolemy/
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

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

\[
\begin{align*}
n & \in \mathcal{N}, \text{the set of numeric, integer literals} \\
c, d & \in \mathcal{C}, \text{a set of class names} \\
t & \in \mathcal{C} \cup \{\text{int}\}, \text{a set of types} \\
p & \in \mathcal{P}, \text{a set of event type names} \\
f & \in \mathcal{F}, \text{a set of field names} \\
m & \in \mathcal{M}, \text{a set of method names} \\
\text{var} & \in \{\text{this}\} \cup \mathcal{V}, \mathcal{V} \text{ is a set of variable names}
\end{align*}
\]
Specification Feature

\[\begin{align*}
contract & ::= \text{requires } sp \text{ assumes } \{ se \} \text{ ensures } sp \\
\text{spec} & ::= \text{requires } sp \text{ ensures } sp \\
sp & ::= n \mid \text{var} \mid sp.f \mid sp \neq \text{null} \mid sp == n \\
& \quad \mid sp == \text{old}(sp) \mid ! sp \mid sp \&\& sp \\
& \quad \mid sp < n \\
se & ::= sp \mid \text{null} \mid \text{new } c() \mid se.m(se*) \mid se.f \mid se.f = se \\
& \quad \mid \text{if } (sp) \{ se \} \text{ else } \{ se \} \mid \text{while } (sp) \{ se \} \\
& \quad \mid \text{cast } c se \mid \text{form } = se; se|se; se \\
& \quad \mid \text{register } (se) \mid \text{invoke } (se) \mid \text{announce } p(e*) \{ e \} \\
& \quad \mid \text{next} \mid \text{spec} \mid \text{either } \{ se \} \text{ or } \{ se \}
\end{align*}\]

\textbf{Figure:} Syntax for writing translucid contracts