Translucid Contracts for Aspect-oriented Interfaces

9th Workshop on Foundations of Aspect-Oriented Languages (FOAL ’10)
Overview Modular Reasoning & Pointcut Fragility for AO Programs

- Many proposals to solve these problems
- OM [Aldrich’05], AAI [Kiczales & Mezini ’05], XPI [Sullivan et al.’05,’09], Event Types [Rajan and Leavens’08], etc.
- Common theme: We need AO interfaces
- AO interfaces solve fragility problems . . .
- . . . and allow writing contracts between base and aspects

Behavioral Contracts Insufficient for AO Interfaces

- Specification of advice input/output isn’t enough
- Need access to internal states that cause control effects

Translucid Contracts: Grey Box Specification

- Provide access to some internal states . . .
- . . . so we can understand and enforce control effects
Outline Explain Translucid Contracts via a Candidate AO Interface

- Quantified, Typed Events [Rajan and Leavens’08]
- Brief background on Ptolemy
- Translucid contracts in Ptolemy

Discuss Properties of Translucid Contracts

- Focus on control flow effects
- Illustrate via Rinard et al.’s classification
- . . . and beyond

Applicability of Translucid Contracts

- Open Modules and XPI (other ideas in paper)
Ptolemy via an Example: Declaring an Event Type

```java
Fig event Changed {
    Fig fe;
}
```

Event types act as interfaces.
Ptolemy via an Example: Announcing an Event

```java
1 class Fig {
2     Fig event Changed{
3         Fig fe;
4     }
5 }
6 class Point{
7     int x; int y;
8     Fig setX(int x) {
9         announce Changed(this) {
10            this.x = x; this
11         }
12     }
13 }
```

- Point is a subject in Ptolemy (II) terminology.
- Subjects are only aware of event types.
- Subjects can be compiled with just event types.

http://www.cs.iastate.edu/~ptolemy/
Ptolemy via an Example: Advising an Event

```
1 Fig event Changed{
2   Fig fe;
3 }
```

```
13 class Update {
14   when Changed do update;
15   Update init()
16     register(this)
17 }
18 Display d;
19 Fig update(thunk Fig rest, Fig fe){
20   d.update(fe);
21   invoke(rest)
22 }
23 }
```

- Update is a handler in Ptolemy (II) terminology.
- Handlers are only aware of event types.
- Handlers can be compiled with just event types.

http://www.cs.iastate.edu/~ptolemy/
Adding Behavioral Contracts to AO Interfaces

```java
Fig event Changed {
  Fig fe;
  requires fe != null
  ensures fe != null
}
```

- Advantage of AO Interfaces: can specify contracts
- Sullivan et al. [XPI ’05,’09] show how to do that
- Specify precondition of event announcement
- Specify postcondition that a handler must ensure

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

Adding Behavioral Contracts to AO Interfaces

```java
class Update {
    Fig update(thunk Fig rest, Fig fe) {
        d.update(fe);
        // Quiz: what is missing here?
    }
}
```

What is the effect of missing code on `Point.setX()`?

http://www.cs.iastate.edu/~ptolemy/
Problems with Behavioral Contracts

1. Fig event
2. Changed {
3. Fig fe;
4. requires
5. fe != null
6. ensures
7. fe != null
8. }
9. class Update {
10. /**
11. ...
12. */
13. Fig update(
14. thunk Fig rest,
15. Fig fe)
16. d.update(fe);
17. //Answer:
18. invoke (rest)
19. }
20. }

Insufficient for:
- Understanding and enforcing control effects
- Reasoning about effects of aspects on each other

http://www.cs.iastate.edu/~ptolemy/9

Translucid Contracts for Aspect-oriented Interfaces
Based on grey box specification [Büchi & Weck ’99]

- **requires** describes precondition of
  - event announcement and **invoke** expressions

- **ensures** describes postcondition of
  - event announcement and **invoke** expressions

- **assumes** block describes behavior of the handlers
A Closer Look at Assumes

/* Contract */
requires fe != null
assumes {
  preserves
    fe == old(fe);

  invoke(next)
}
ensures fe != null

*/ Handler Method */
Fig update(thunk Fig rest,
            Fig fe){
  refining preserves
    fe==old(fe){
      d.update(fe);
    }
    invoke(rest)
}
Handler Verification Step I (Details in our Report)

- Each handler for event $p$ must match \texttt{assumes} block of $p$
- Checking this requires handler code and the event type $p$
- Thus, this step is modular

/* Contract */

\begin{verbatim}
requires \ fe != \null
assumes { 
  preserves 
  \fe == \old(\fe);

  invoke(next)
}

ensures \ fe != \null
\end{verbatim}

/* Handler Method */

\begin{verbatim}
Fig update(thunk Fig rest, Fig fe) {
  refining preserves 
  \fe==\old(\fe) {
    d.update(\fe);
  }

  invoke(rest)
}
\end{verbatim}
Handler Verification Step II (Details in our Report)

- Replace (lazily) each `invoke` in handler method by:

```java
    either {
      requires fe!=null ensures fe!=null
    }
    or {
      preserves fe==old(fe) ;
      invoke(rest)
    }
```

- and apply weakest precondition-based reasoning.
- This also requires only handler code and the event type `p`
- Thus, this step is modular also
Subject Verification (Details in our Report)

► Replace (lazily) each `announce` by:

```java
1   either {
2     requires fe!=null;
3     this.x = x; this
4     ensures fe!=null
5 }
6  or {
7    preserves this==old(this) ;
8    invoke(rest)
9 }
```

► and apply weakest precondition-based reasoning.
► This also requires only subject code and the event type `p`
► Thus, this step is **modular** also
Outline for Rest of the Talk

- Analyze our proposal from two different perspectives
- Expressiveness: what kinds of control effects can we specify?
  - Rinard et al.’s classification [FSE ’04]
  - augmentation, replacement, narrowing, combination
  - Properties beyond this classification
- Applicability: is our idea limited to Ptolemy?
  - Apply it to other AO interfaces
  - XPI [Sullivan et al ’05, ’09]
  - AAI [Kiczales & Mezini ’05]
  - Open Modules [Aldrich ’05]
Event Type Permitting After Augmentation

```plaintext
1 Fig event Changed {
2   Fig fe;

3 requires fe != null
4 assumes {
5       invoke(next);
6       preserves fe==old(fe)
7   }
8 ensures fe != null
9 }
```

- Similar to before augmentation.
- Handler must run exactly one invoke.

http://www.cs.iastate.edu/~ptolemy/
Event Type Permitting Narrowing **Handlers are allowed to not invoke under certain conditions**

```java
class Fig { int fixed; }
Fig event Changed {
    Fig fe;
    requires fe != null
    assumes {
        if(fe.fixed == 0) { invoke(next); }
        else { preserves fe==old(fe) }
    }
    ensures fe != null
}
```

- Illustrates use of conditionals in contract
- Only the event’s context variable may be named in the `assumes` block of that event
Event Type Permitting Replacement **Handlers do not invoke, thus they replace event body**

```java
1 Fig event Moved {
2     Point p;
3     int d;
4     requires p!=null && d>0
5     assumes {
6         preserves p!=null && p.y == old(p.y)
7     }
8     ensures p!=null
9 }
```

- If there is no `invoke` in the `assumes` block then a handler may not invoke
Event Type Permitting Combination Handlers may invoke multiple times

```java
1    assumes {
2        while (fe.colorFixed==0) {
3           // ...
4           invoke (next);
5           // ...
6        }
7    }
```

- Conforming handlers must have a loop at the same position for the structure to match
- The test condition of loop must match also
Beyond Rinard’s Control Flow Properties

```
1 class Point extends Fig{
2   int x; int y; int s;
3   Point init(int x, int y){
4     this.x=x; this.y=y;
5     this.s=1; this }
6   int getX(){this.x*this.s}
7   int getY(){this.y*this.s}
8   Fig move(int x, int y){
9     announce Moved(this){
10      this.x=x;this.y=y; this }
11   Fig event Moved{
12      Point p;
13      requires p!=null
14      assumes{
15        invoke(next);
16        if(p.x<5&& p.y<5){
17            establishes p.s==10
18        } else {establishes p.s==1}
19      ensures p!=null
20   class Scaling {
21      when Moved do scale;
22      Fig scale(thunk Fig rest,
23                 Point p){
24        invoke(rest);
25        if(p.x<5 && p.y<5){
26            refining establishes p.s==10{
27                p.s=10; p
28            }
29        } else {
30            refining establishes p.s==1{
31                p.s == 1; p }}}}
```
Cross-Cutting Interface (XPI) AAI is just XPI, details in the paper.
Open Modules

module FigModule {
  class Fig;
  friend Enforce;
  expose:
    target(fe) && call(
      void Fig+.set*(..));

  requires fe != null
  assumes {
    if(fe.fixed == 0){
      proceed()
    } else {
      preserves
      fe == old(fe)}
  }
  ensures fe! = null
}

aspect Enforce {
  Fig around (Fig fe): target
    && call(void Fig+.set*(..)){
      if(fe.fixed == 0){
        proceed()
      } else {
        refining preserves
        fe==old(fe) {
          fe
        }
      }
    }
}
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]

- First to consider grey box specification to enable modular reasoning about code that announces events from the code that handles events
Translucid Contracts for Expressive Specification

**Broad Problem: Modular reasoning and pointcut fragility**
- Aspect-oriented interfaces solve part of it
- e.g, XPI, AAI, OM, etc
- Mostly solve pointcut fragility problem

**Specific Problem: Reason about Modules in Isolation**
- Typically, AO interfaces annotated with behavioral contract
- Specify relation between module’s input and output
- But can not reveal internal states

**Solution: Translucid Contracts**
- Expressive specification
- Allows modular verification of control effects
- Show applicability to other AO interfaces
Translucid Contracts for Aspect-oriented Interfaces

Acknowledgments

Thanks to: the US National Science Foundation, Kevin J. Sullivan, William G. Griswold, Robert Dyer, Juri Memmert, and anonymous reviewers for the FOAL workshop.

ptolemyj.sourceforge.net

http://www.cs.iastate.edu/~ptolemy/
Ptolemy’s Syntax

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

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

\[
\text{contract ::= requires } sp \text{ assumes } \{ \text{se} \} \text{ ensures } sp
\]
\[
\text{spec ::= requires } sp \text{ ensures } sp
\]
\[
sp ::= n \mid \text{var} \mid sp.f \mid sp \neq \text{null} \mid sp == n
\]
\[
| sp == \text{old}(sp) \mid \neg sp \mid sp \& \& sp
\]
\[
| sp < n
\]

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

**Figure:** Syntax for writing translucid contracts