Translucid Contracts for Aspect-oriented Interfaces

9th Workshop on Foundations of Aspect-Oriented Languages (FOAL ’10)
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)
Event types act as interfaces.
Ptolemy via an Example: Announcing an Event

Fig

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

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

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

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

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

▶ Update is a handler in Ptolemy (II) terminology.
▶ Handlers are only aware of event types.
▶ Handlers can be compiled with just event types.
Adding Behavioral Contracts to AO Interfaces

```java
1 Fig event Changed {
2   Fig fe;
3   requires fe != null
4   ensures fe != null
5 }
```

- 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/
What is the effect of missing code on Point.setX()?
Problems with Behavioral Contracts

1 Fig event Changed {
2 Fig fe;
3 requires fe != null
4 ensures fe != null
5 }

13 class Update {
14 /**
15 * ...
16 */
17 Fig update(thunk Fig rest, Fig fe){
18   d.update(fe);
19   //Answer: invoke(rest)
20 }  
21 }

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

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

- 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

```c
/* Contract */
requires fe != null
assumes {
    preserves
    fe == old(fe);
    invoke(next)
}
ensures fe != null

▶ assumes shows parts of a handler and hides the rest
▶ Hiding is done using specification expressions
▶ All invoke expressions are explicit
```

```c
/* Handler Method */
Fig update(thunk Fig rest,
            Fig fe) {
    refining preserves
    fe==old(fe) {
        d.update(fe);
    };
    invoke(rest)
}
```

http://www.cs.iastate.edu/~ptolemy/
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 \textbf{modular}.

\begin{verbatim}
/* 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)
}
\end{verbatim}

\url{http://www.cs.iastate.edu/~ptolemy/}

Translucid Contracts for Aspect-oriented Interfaces
Handler Verification Step II (Details in our Report)

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

  ```
  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.
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.
Handlers are allowed to not invoke under certain conditions

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

- Illustrates use of conditionals in contract
- Only the event’s context variable may be named in the `assumes` block of that event

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

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

```java
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 }}}}
```

http://www.cs.iastate.edu/~ptolemy/
Cross-Cutting Interface (XPI)

AAI is just XPI, details in the paper.
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

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

Join us at...

ptolemyj.sourceforge.net
Ptolemy’s Syntax

\[
\begin{align*}
\text{prog} &::= \text{decl}^*\ e \\
\text{decl} &::= \text{class}\ c\ \text{extends}\ \text{d}\ \{\ \text{field}^*\text{meth}^*\ \text{binding}^*\ \} \\
&\quad|\ t\ \text{event}\ p\ \{\ \text{form}^*\ \text{contract}\ \}
\end{align*}
\]

\[
\begin{align*}
\text{field} &::= t\ f; \\
\text{meth} &::= t\ m\ (\text{form}^*)\ \{\ e\ \}\ |\ t\ m\ (\text{thunk}\ t\ \text{var},\ \text{form}^*)\ \{\ e\ \}
\end{align*}
\]

\[
\begin{align*}
\text{form} &::= t\ \text{var},\ \text{where}\ \text{var} \neq \text{this} \\
\text{binding} &::= \text{when}\ p\ \text{do}\ m
\end{align*}
\]

\[
\begin{align*}
\text{e} &::= n\ |\ \text{var}\ |\ \text{null}\ |\ \text{new}\ c()\ |\ e.m(e^*)\ | e.f\ | e.f = e \\
&\quad|\ \text{if}\ (e_p)\ \{\ e\ \}\ \text{else}\ \{\ e\ \}\ |\ \text{while}\ (e_p)\ \{\ e\ \}\ |\ \text{cast}\ c\ e \\
&\quad|\ \text{form} = e;\ e\ |\ e;\ e|\ \text{register}(e)\ |\ \text{invoke}(e) \\
&\quad|\ \text{announce}\ p\ (e^*)\ \{\ e\ \}\ |\ \text{refining}\ \text{spec}\ \{\ e\ \}
\end{align*}
\]

\[
\begin{align*}
\text{ep} &::= n\ |\ \text{var}\ |\ e_p.f|\ e_p \neq \text{null}\ |\ e_p \neq n\ |\ !\ e_p\ |\ e_p \&\&\ e_p
\end{align*}
\]

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


**Specification Feature**

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

**Figure:** Syntax for writing translucid contracts