

Mehdi Bagherzadeh
Iowa State Univeristy
mbagherz@cs.iastate.edu

Hridesh Rajan
Iowa State University
hridesh@cs.iastate.edu

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

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

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

Event types act as interfaces.

Ptolemy via an Example: Announcing an Event

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

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

- 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

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

```
class Update {
 when Changed do update;
 Update init(){
  register(this)
 }
 Display d;
 Fig update(thunk Fig rest,
                 Fig fe){
  d.update(fe);
  invoke(rest)
 }}
```

- 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

```
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

Translucid Contracts

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

- 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)
 }
```

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

Handler Verification Step I (Details in our Report)

- Each handler for event ***p*** must match **`assumes`** block of ***p***
- Checking this requires handler code and the event type ***p***
- Thus, this step is <u>modular</u>

```
/* 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 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 <u>modular</u> also

Subject Verification (Details in our Report)

- Replace (lazily) each **announce** by:

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

- 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

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

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

Event Type Permitting Narrowing **Handlers are allowed to not invoke under certain conditions**

```
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**

```
Fig event Moved {
 Point p;
 int d;
 requires p!=null && d>0
 assumes {
  preserves p!=null && p.y == old(p.y)
 }
 ensures p!=null
}
```

- If there is no **`invoke`** in the **`assumes`** block then a handler may not invoke

Event Type Permitting Combination **Handlers may invoke multiple times**

```
assumes {
 while(fe.colorFixed==0){
  // ...
  invoke(next);
  // ...
 }
}
```

- 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

```
class Point extends Fig{
 int x; int y; int s;
 Point init(int x,int y){
  this.x=x; this.y=y;
  this.s=1; this }
 int getX(){this.x*this.s}
 int getY(){this.y*this.s}
 Fig move(int x, int y){
  announce Moved(this){
   this.x=x;this.y=y; this }}}
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}
```

```
class Scaling {
 when Moved do scale;
 Fig scale(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 }}}}
```

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): targe
  && 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

Ptolemy's Syntax

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

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

Specification Feature

$$
\begin{array}{lll}
\textit{contract} & ::= & \textbf{requires}\ sp\ \textbf{assumes}\ \{\ se\ \}\ \textbf{ensures}\ sp \\
\textit{spec} & ::= & \textbf{requires}\ sp\ \textbf{ensures}\ sp \\
sp & ::= & n \mid var \mid sp.f \mid sp\ \texttt{!=}\ \textbf{null} \mid sp == n \\
 & & \mid sp == \textbf{old}(sp) \mid\ !\ sp \mid sp\ \texttt{\&\&}\ sp \\
 & & \mid sp < n
\end{array}
$$

$$
\begin{array}{ll}
se ::= & sp \mid \textbf{null} \mid \textbf{new}\ c() \mid se.m(\ se^*\ ) \mid se.f \mid se.f = se \\
 & \mid \textbf{if}\ (sp)\ \{\ se\ \}\ \textbf{else}\ \{\ se\ \} \mid \textbf{while}\ (sp)\ \{\ se\ \} \\
 & \mid \textbf{cast}\ c\ se \mid form = se;\ se|se;\ se \\
 & \mid \textbf{register}(\ se\ ) \mid \textbf{invoke}\ (\ se\ ) \mid \textbf{announce}\ p\ (\ e^*\ )\ \{\ e\ \} \\
 & \mid \textbf{next} \mid spec \mid \textbf{either}\ \{\ se\ \}\ \textbf{or}\ \{\ se\ \}
\end{array}
$$

Figure: Syntax for writing translucid contracts