Translucid Contracts
in
The Ptolemy Programming Language

(Expressive Specification & Modular Verification for Aspect-oriented Interfaces)
## 6 Known Problems in AO Literature

<table>
<thead>
<tr>
<th>Problem</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to overcome <strong>pointcut fragility</strong>?</td>
<td>[Tourwé-Brichau-Gybels SPLAT’03, Stoerzer-Graf ICSM’05, ...]</td>
</tr>
<tr>
<td>How to address <strong>quantification failure</strong>?</td>
<td>[Sullivan et al. ESEC/FSE’05, Griswold et al. IEEE Software 2006, ...]</td>
</tr>
<tr>
<td>How to make <strong>context access expressive</strong>?</td>
<td>[Sullivan et al. ESEC/FSE’05, Griswold et al. Software 2006, ...]</td>
</tr>
<tr>
<td>How to limit the number of composition-related verification tasks due to <strong>pervasive join points</strong>?</td>
<td>[Clifton-Leavens’03, Aldrich’05, Dantas-Walker’06, ...]</td>
</tr>
<tr>
<td>How to modularly verify <strong>control effects of aspects</strong>?</td>
<td>[Zhao-Rinard FASE’03, Rinard-Salcianu-Bugrara FSE’04, ...]</td>
</tr>
<tr>
<td>How to modularly verify <strong>heap effects of aspects</strong>?</td>
<td>[Clifton-Leavens FOAL’03, Katz FOAL’04, Krishnamurthi FSE’04, ...]</td>
</tr>
</tbody>
</table>
Fragility & Quantification

 Fragile Pointcuts: consider method “settled”

```plaintext
1 Fig around(Fig fe) : 
2 call(Fig+.set*(..)) && target(fe) 
3 ...
```

 Quantification Failure: Arbitrary events not available

```plaintext
1 Fig setX(int x){
2   if (x == this.x) return;
3   else {
4     this.x = x;
5   }
5 }
```

Inadvertant match with regex-based pointcut

Abstract events often not available at the interface.
Context access

- Limited Access to Context Information
  - Limited reflective interface (e.g. “thisJoinPoint” in AJ)
  - Limited Access to Non-uniform Context Information

```java
1 Fig around(Fig fe) :
2 call(Fig+.set*(..)) && target(fe)
3 || call(Fig+.makeEq*(..)) && args(fe){
4 ...
```

Encoding knowledge about base code in aspect
For each join point shadow, all applicable aspect should be considered (whole-program analysis)
# 6 Known Problems in AO Literature


## Around the same time:

- **EJP**: Hoffman & Eugster’07,
- **IIIA**: Steimann et al.’10

## How to modularly verify control effects of aspects?

[Zhao-Rinard FASE’03, Rinard-Salcianu-Bugrara FSE’04]

## How to modularly verify heap effects of aspects?

[Clifton-Leavens FOAL’03, Katz FOAL’04, Krishnamurthi FSE’04]
Ptolemy’s Design

- Inspired from implicit invocation (II) approaches
  [Field: Reiss`90, II: Garlan-Notkin`91, Rapide: Luckham-Vera`95]

- … as well as from aspect-oriented (AO) approaches
  [HyperJ: Ossher & Tarr, AspectJ: Kiczales et al.`01, Caeser: Mezini & Ostermann`03, Eos: Rajan & Sullivan`03, XPI: Sullivan et al.`05, Griswold et al.`06, OM: Aldrich`05, AAI: Kiczales & Mezini`05]
Ptolemy’s Design Goals

- Enable modularization of crosscutting concerns, while preserving encapsulation of object-oriented code,

- enable well-defined interfaces between object-oriented code and crosscutting code, and

- enable separate type-checking, separate compilation, and modular reasoning of both OO and crosscutting code.
First and foremost

- Main feature is event type declaration.

- Event type declaration design similar to API design.
  - What are the important abstract events in my application?
  - When should such events occur?
  - What info. must be available when such events occur?

- Once you have done it, write an event type declaration.
Type Declaration for Abstract Events

- Event type declaration is an abstraction.
- Declares context available at the concrete events.
Explicit, More Declarative Event Announcements

```java
class Fig {
    bool isFixed;
}

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

- Explicit, more declarative, typed event announcement.
Advising Events

- No special type of “aspect” modules
- Unified model from Eos [Rajan and Sullivan 2005]

```java
class Enforce {
    @Register
    Enforce() {
    }
}
```
Quantification Using Binding Decls.

- Binding declarations
  - Separate “what” from “when” [Eos 2003]

```java
class Enforce {
    @Register
    Enforce(){}

    Fig enforce(Changed next){
        ...
    }

    when Changed do enforce;
}
```
Controlling Overriding

- Use `invoke` to run the continuation of an event
- Allows overriding similar to AspectJ

```java
class Enforce {
    @Register
    Enforce(){};

    Fig enforce(Changed next) {
        if (!next.fe.isFixed)
            return invoke(next);
        else
            return next.fe;
    }

    when Changed do enforce;
}
```
Ptolemy Example: All Together

- Skip the execution of `setX()` when `isFixed` is true.

- Event-driven-programming:
  - Subject `Point` announces event `Changed` when `setX()` is called.
  - Event handler `enforce` registers for `Changed` and runs upon its announcement.
  - Handler `enforce` implements the example requirement

- … also supports mixin-like inter-type declarations [Bracha & Cook]
6 Known Problems in AO Literature

Since these definitions differ ...

**BASIC DEFINITIONS**
When is separation of crosscutting concerns accomplished?

- Scattered and tangled concerns are textually separated,

  +

- One can modularly verify module-level properties of separated concerns.
When is a verification task “modular”?  

- If it can be carried out using:
  - the code in question
  - specifications of static types mentioned in code.
Understanding Control Effects

- **Logging** & **Enforce** advise the same set of events, **Changed**

- Control effects of both should be understood when reasoning about the base code which announces **Changed**

```c
21 class Enforce {  
22   ...  
23   Fig enforce(Changed next){  
24     if(!next.fe.isFixed)  
25       return invoke(next)  
26     else  
27       return next.fe;  
28   }  
29   when Changed do enforce;  
30 }
```

```c
31 class Logging{  
32   ...  
33   Fig log(Changed next){  
34     if(!next.fe.isFixed)  
35       return invoke(next);  
36     else {  
37       Log.log(next.fe); return next.fe;  
38       }  
39   when Changed do log;  
40 }
```
Can Specifications help?

```plaintext
10 Fig event Changed {
11 Fig fe;
12 \textbf{requires} fe \neq \text{null}
13
14
15
16
17
18 \textbf{ensures} fe \neq \text{null}
20 }

21 \textbf{class} Enforce {
22  ...
23  Fig enforce(Changed next) {
24    \textbf{if}(!next.fe.isFixed)
25    \textbf{return} invoke(next)
26  \textbf{else}
27    \textbf{return} next.fe;
28  }
29 }
30
31 \textbf{class} Logging{
32  ...
33  Fig log(Changed next) {
34    \textbf{if}(!next.fe.isFixed)
35    return invoke(next);
36  \textbf{else} {
37    Log.log(next.fe); \textbf{return} next.fe;
38  }
39  \textbf{when} Changed \textbf{do} log;
40 }
```
Blackbox Can’t Specify Control

- Blackbox can't specify properties like “advice must proceed to the original join point”.
- If invoke goes missing, then execution of Logging is skipped.
  - Ptolemy’s invoke = AspectJ’s proceed
Blackbox Can’t Specify Composition

- Different orders of composition may result in different control flow if `invoke` is missing
  - Logging runs first, Enforce is executed
  - Enforce runs first, Logging is skipped
Translucid Contracts (TCs)

- TCs enable specification of control effects

- Greybox-based specification [Büchi and Weck `99]
  - Hides some implementation details
  - Reveals some others

- Limits the behavior & structure of aspects applied to AO interfaces
Translucid Contracts Example

- Limits the behavior of the handler
  - **requires/ensures** labels pre/postconditions
- Greybox limits the handler’s code
  - **assumes** block with program/spec. exprs
Assumes Block

A mixture of

- **Specification** exprs
  - Hide implementation details

- **Program** exprs
  - Reveal implementation details
TCs Can Specify Control

1. TC specifies control effects independent of the implementation of the handlers Enforce, Logging, etc.

2. `invoke(next)` in TC assures `invoke(next)` in `enforce` cannot go missing.
   - Proceeding to the original join point is thus guaranteed.

3. Different orders of composition of handlers doesn’t result in different control flow.
Modular Verification of Ptolemy Programs

1. Verifying that a handler refines the contract of the event it handles.
   - Verified modularly

2. Verifying code containing `announce/invoke` exprs.
   - which cause unknown set of handlers to run.
   - Verified modularly

Translucid contracts enable modular verification of control effects.
Handler Refinement

- A handler structurally matches the `assumes` block of the TC of the event it handles.
  - Structural refinement
  - Statically, during type-checking

- A handler respects pre/postconditions of the `requires/ensures` predicate in TC.
  - Dynamically, using runtime assertion checks
Handler Refinement

- Structural refinement:
  - A program expr. is refined by a textually matching prog. expr.
  - A specification expr. is refined by a refining expr. with the same spec.
  - Structural refinement is done statically at type-checking phase.

- TC’s Pre-/postconditions are enforced using runtime assertion checks (RACs)
Verification of Announce & Invoke

• Announce & Invoke cause, unknown set of handlers to run.

• Translucid contracts, provide a sound spec. of the behavior for an arbitrary number of handlers.

• Translation function, Tr, computes the specification.
Verification of Announce, Subject Code

Apply $\mathsf{Tr}$ to the code containing announce:

- $\mathsf{Tr}$ replaces announce with a spec representing situations when there are:
  - No More handlers to run
    - Event body is executed
  - More handler to run
    - Next handler is executed.
    Translation of the TC is the spec of the running handler
Example of Verification of Announce, Subject Code

- Replace `announce` by the spec. computed by `Tr` function.
- either branch: no more handlers to run: event body + parameters
- Or branch: more handlers to run: apply `Tr` to TC’s assumes block
Verification of Announce & Invoke, Similarities & Differences

Apply \( \text{Tr} \) to the code containing \texttt{announce/invoke}:

\( \text{Tr} \) replaces announce/invoke with a spec representing situation where there are:

- **No More** handler to run
  - **Announce**: Event body is executed and is accessible.
  - **Invoke**: Event body is executed but not accessible.
    TC’s requires/ensures represent the event body.

- **More** handlers to run
  - **Announce/invoke**: Next handler is executed.
    Translation of the TC is the spec of the running handler
Runtime Assertion Checking (RAC)

RACs are used to enforce:

- Requires/ensures predicates of the TC, at:
  - beginning/end of each refining handler.
  - before/after invoke exprs.
  - before/after announce exprs.
  - beginning/end of event body.

- Spec. of the refining exprs, at:
  - beginning/end the refining expr. block
Expressiveness of TCs

- All categories of Rinard’s control interference & beyond are expressible using TCs

- Rinard’s control interference categories are concerned about:
  - Number of invoke (proceed) exprs in the handler (advice)
  - Details in paper.
Related Ideas

 Contracts for Aspects
   XPI [Sullivan et al. `09], Cona [Skotiniotis & Lorenz `04], Pipa [Zhao & Rinard `03]
    ➢ XPI’s contracts informal, all blackbox contracts

 Modular reasoning for Aspects
   [Krishnamurthi, Fishler, Greenburg `04]
    ➢ Blackbox contracts, global pre-reasoning step
   [Khatchadourian-Dovland-Soundarajan `08]
    ➢ Blackbox contracts, additional pre-reasoning step to generate traces.

 Effective Advice [Oliveira et al. `10]
   No quantification
Conclusion & Future Work


Translucid Contracts

How to modularly verify heap effects of aspects?
[Clifton-Leavens FOAL’03, Katz FOAL’04, Krishnamurthi FSE’04]
Translucid Contracts
in
The Ptolemy Programming Language

(Expressive Specification & Modular Verification for Aspect-oriented Interfaces)