Modularizing Crosscutting Concerns with Ptolemy

Hridesh Rajan

with Gary T. Leavens, Sean Mooney, Robert Dyer, and Mehdi Bagherzadeh

Twitter: @ptolemyj

This work is supported by the US National Science Foundation (NSF)
under grants CCF-10-17334 and 10-17262.

Outline

- Why Ptolemy? What problems does it solve?
  - Two precursors
    - Implicit Invocation and Aspect-orientation
- Ptolemy and how it solves these problems.
- Main Language Features
  - Declarative, typed events (join points in AO terms)
  - Declarative, typed event announcement (no AO term)
  - Declarative, typed event registration (advising in AO terms)
  - Quantification based on event types (same as the AO term)

Outline

- Modular Verification Features
  - Translucid Contracts (no AO term)

    [Also in the main conference: Thursday @ 11 AM]
- Where to use Ptolemy Features?
  - vs. Aspect-orientation,
  - vs. Implicit Invocation
- State of Tools
- Opportunities to Contribute
- Conclusion

One shall not have to choose between reasoning and separation.

WHY PTOLEMY?

Need for Improved Separation

- Some concerns hard to modularize
- Number of proposals: Units [Flatt and Felleisen], Mixin [Bracha and Cook], Open Classes [Clifton et al.], Roles [Kristensen and Osterbye], Traits [Scharli et al.], Implicit Invocation [Garlan, Notkin, Sullivan et al.], Hyperslices [Ossher and Tarr], Aspects [Kiczales et al.], etc
- Shows that there is a real need

Two similar ideas

- Implicit invocation (II) vs. Aspect-orientation (AO)
- … both effective for separation of concerns
- … both criticized for making reasoning hard
  - II criticized in early/late 90's
  - AO criticized in early 2000's

- Ptolemy is designed to
  - combine best ideas from II and AO
  - … and to make reasoning easier

[JHotDraw – Gamma et al.]

RUNNING EXAMPLE

Elements of a Drawing Editor

- Elements of drawing
  - Points, Lines, etc
  - All such elements are of type Fig
- Challenge I: Modularize display update policy
  - Whenever an element of drawing changes — Update the display
- Challenge II: Impose application-wide restriction
  - No element may move up by more than 100

Figure Elements

```
abstract class Fig {
}
```

- Fig – super type for all figure elements
  - e.g. points, lines, squares, triangles, circles, etc.

Point and its Two Events

```
class Point extends Fig {
  int x;
  int y;
  void setX(int x) {
   this.x = x;
  }
  ..
  void makeEqual(Point other) {
   if(!other.equals(this)) {
     other.x = this.x;
     other.y = this.y;
  }}}
```

- Changing Fig is different for two cases.
- Actual abstract event inside makeEqual is the true branch.

Reiss'92, Garlan and Notkin'92

IMPLICIT INVOCATION

Key Ideas in II

- Allow management of name dependence
  - when "Point's coordinates changes" update Display
  - ... but Point shouldn't depend on Display
  - ... complicates compilation, test, use, etc
- Components (subjects) declare events
  - e.g. when "Point's coordinates changes"
  - provide mechanisms for registration
  - ... and for announcement
- Components (observers) register with events
  - e.g. invoke me when "Point's coordinates changes"
- Subjects announce events
  - e.g. when "Point's coordinates changes"
  - "change in coordinates" event announced

II: Components Declare Events

```
abstract class Fig {
    List changeObservers;
    void announceChangeEvent(Fig changedFE){
      for(ChangeObserver o : changeObservers){
        o.notify(changedFE);
      }
    }
    void registerWithChangeEvent(ChangeObserver o){
      changeObservers.add(o);
   }
}
abstract class ChangeObserver {
  void notify(Fig changedFE);
}
```

II: Components Announce Events

```
class Point extends Fig {
  int x; int y;
  void setX(int x) {
    this.x = x;
    announceChangeEvent(this);
  }
  void makeEqual(Point other) {
    other.x = this.x; other.y = this.y;
    announceChangeEvent(other);
  }
}
```

- Event announcement explicit, helps in understanding
- Event announcement flexible, can expose arbitrary points

II: Component Register With Events

```
class Update extends ChangeObserver {
   Fig last;
   void registerWith(Fig fe) {
     fe.registerWithChangeEvent(this);
   }
   void notify(Fig changedFE){
     this.last = changedFE;
     Display.update();
   }
}
```

- Registration explicit and dynamic, gives flexibility
- Generally deregistration is also available

II: Disadvantages

- Coupling of observers to subjects

```
void registerWith(Fig fe) {
   fe.registerWithChangeEvent(this); ...
}
```

- Lack of quantification

```
void registerWith(Point p){
   p.registerWithChangeEvent(this);
}
void registerWith(Line l) {
   l.registerWithChangeEvent(this);
}
```

II: Disadvantages

- No ability to replace event code

```
class MoveUpCheck extends … {
    void notify(Fig targetFE, int y, int delta) {
      if (delta < 100) { return targetFE }
     else{throw new IllegalArgumentException()}
    }
  }
```

Kiczales et al. 97, Kiczales et al. 2001

ASPECT-BASED SOLUTIONS

Key Similarities/Differences with II

- Events ≡ "join points"
  - AO: pre-defined by the language/ II: programmer
  - AO: Implicit announcement/ II: explicit
- Registration ≡ Pointcut descriptions (PCDs)
  - AO: declarative
- Handlers ≡ "advice" register with sets of events
- Quantification: using PCDs to register a handler with an entire set of events

Aspect-based Solution

```
aspect Update {
Fig around(Fig fe) :
    call(Fig+.set*(..)) && target(fe)
   || call(Fig+.makeEq*(..)) && args(fe){
   Fig res = proceed(fe);
   Display.update();
   return res;
}
```

Advantages over II

- Ease of use due to quantification
- By not referring to the names, handler code remains syntactically independent

Limitations: Fragility & Quantification

- Fragile Pointcuts: consider method "settled"

```
Fig around(Fig fe) :
call(Fig+.set*(..)) && target(fe)
|| call(Fig+.makeEq*(..)) && args(fe){
...
```

- Quantification Failure: Arbitrary events not available

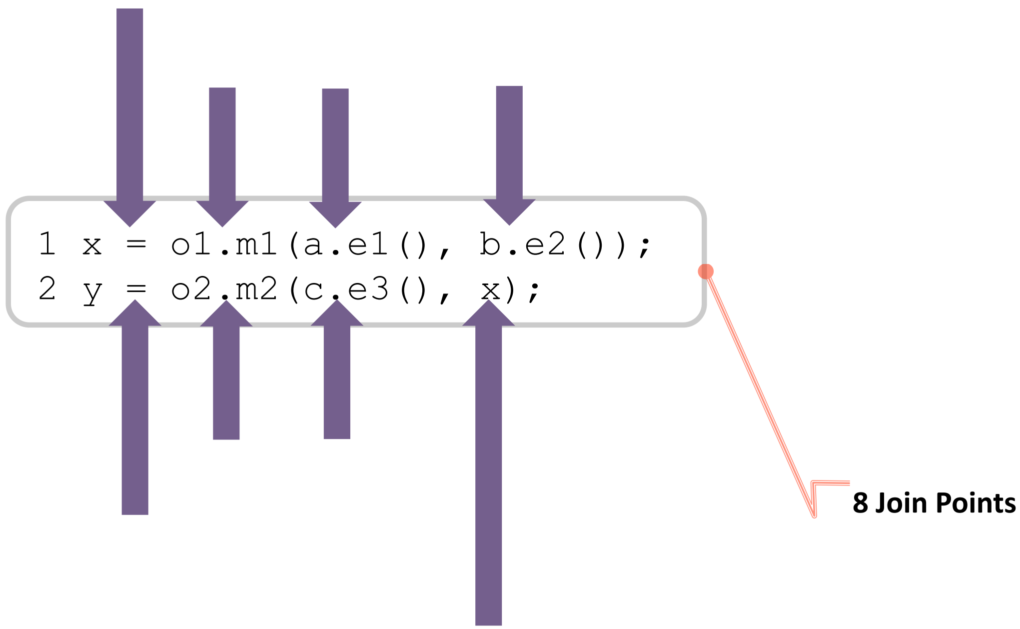
```
Fig setX(int x){
  if (x.eq(this.x)) { return this; }
  /* abstract event change */
  else { this.x = x; return this; }
}
```

Limitations: Context access

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

```
Fig around(Fig fe) :
call(Fig+.set*(..)) && target(fe)
|| call(Fig+.makeEq*(..)) && args(fe){
...
```

Limitations: Pervasive Join Point Shadows

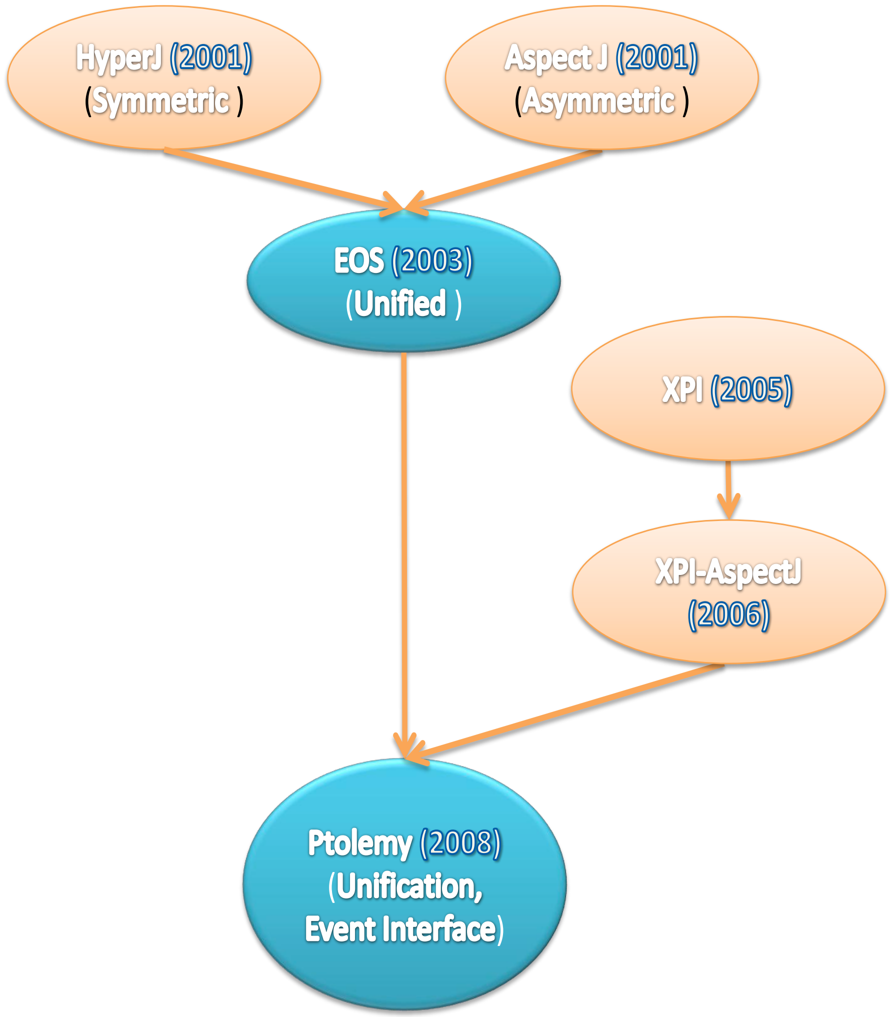


- For each join point shadow, all applicable aspect should be considered (whole-program analysis)

Ptolemy (Claudius Ptolemaeus), fl. 2d cent. A.D., celebrated Greco-Egyptian mathematician, astronomer, and geographer.

THE PTOLEMY LANGUAGE

Evolution of the Ptolemy Language



Design Goals of Ptolemy

- 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.

Declaring an Event Type

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

Event Type Declaration

Declaring an Event Type

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

Event Type Declaration

- Event type is an abstraction.
- Declares context available at the concrete events.
- Interface, so allows design by contract (DBC) methodology.

Announcing Events in Ptolemy

Subject

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

Event Announcement

- Explicit, more declarative, typed event announcement.

More Event Announcements

Subject

```
class Point extends Fig{
..
   Fig moveUp(int delta){
    announce MoveUpEvent(this){
     this.y += delta; return this;
    }
   }
}
```

Event Announcement

- Explicit, more declarative, typed event announcement.

Advising Events

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

Observer(Handler)

```
class DisplayUpdate {


}
```

Quantification Using Binding Decls.

- Binding declarations
  - Separate "what" from "when" [Eos 2003]

Observer(Handler)

```
class DisplayUpdate {

  when Changed do update;
}
```

Quantification

Dynamic Registration

- Allow dynamic registration
  - Other models can be programmed

Observer(Handler)

```
class DisplayUpdate {

 void DisplayUpdate(){ register(this)}

 Fig update(Changed next){


 }

 when Changed do update;
}
```

Registration

Quantification

Controlling Overriding

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

Observer (Handler)

```
class DisplayUpdate {

 void DisplayUpdate(){ register(this)}

 Fig update(Changed next){
      invoke(next);
      Display.update();
      System.out.println("After Invoke");
 }

 when Changed do update;
}
```

Registration

Running continuation of the event

Quantification

Exercise 0: Get the distribution

- Go to the URL to download Ptolemy1.2 Beta1

  http://www.cs.iastate.edu/~ptolemy/aosd11

  and download the zip file ***ptolemy-aosd-11.zip***

- Unzip the contents at a convenient location.

Exercise 1: Figure Editor Example

- [a]Open event type def. in FEChanged.java
  - Note return type and context variables

- [b]Open file Point.java
  - Note event announcements in `setX`, `setY,moveBy`
  - Is the context being passed correctly in `makeEqual`?

Exercise 1: Figure Editor Example

- [c]Open file DisplayUpdate.java
  - Note the annotation form of binding declarations
    - `@When(FEChanged.class)`
    - Sugar for "`when FEChanged do handler;`"
  - Note the annotation form of Register statements
    - `@Register`
    - It registers the receiver object to listen to events mentioned in the binding declarations
    - It is also a sugar for `register(this)`

Enabling modular verification

CONTRACTS IN PTOLEMY

Understanding Control Effects

```
class Enforce {
 …
 Fig enforce(Changed next){
  if(!next.fe.isFixed)
   invoke(next)
  else
    return fe;
 }
 when Changed do enforce;
}
```

```
class Logging{
 …
 Fig log(Changed next){
  if(!next.fe.isFixed)
   invoke(rest);
  else {
   Log.logChanges(fe); return fe;
 }}
 when Changed do log;
}
```

- 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

Blackbox Can't Specify Control

```
Fig event Changed {
 Fig fe;
 requires fe != null






 ensures fe != null
}
```

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

```
class Logging{
 …
 Fig log(Changed next){
  if(!next.fe.isFixed)
   invoke(rest);
  else {
   Log.logChanges(fe); return fe;
  }}
 when Changed do log;
}
```

- Blackbox isn't able to specify properties like advice proceeding 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

```
class Enforce {
 …
 Fig enforce(Changed next){
  if(!next.fe.isFixed)
   invoke(next)
  else
    return fe;
 }
 when Changed do enforce;
}
```

```
class Logging{
 …
 Fig log(Changed next){
   if(!next.fe.isFixed)
    invoke(rest);
   else {
    Log.logChanges(fe); return fe;
  }}
 when Changed do log;
}
```

- Different orders of composition results in different outcomes if invoke is missing
  - Logging runs first, Enforce is <u>executed</u>
  - Enforce runs first, Logging is <u>skipped</u>

Translucid Contracts (TCs)

- TCs enable specification of control effects
- Greybox-based specification
  - Hides some implementation details
  - Reveals some others
- Limits the behavior & structure of aspects applied to AO interfaces

Translucid Contracts Example

```
Fig event Changed {
 Fig fe;
 requires fe != null
 assumes{
  if(!fe.isFixed)
   invoke(next)
  else
   establishes fe==old(fe)
 }
 ensures  fe != null
}
```

Translucid Contract

- 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

```
Fig event Changed {
 Fig fe;
 requires fe != null
 assumes{
  if(!fe.isFixed)
   invoke(next)
  else
   establishes fe==old(fe)
 }
 ensures  fe != null
}
```

- A mixture of
  - Specification exprs
    - Hide implementation details
  - Program exprs
    - Reveal implementation details

TCs Can Specify Control

```
Fig event Changed {
 Fig fe;
 requires fe != null
 assumes{
  if(!fe.isFixed)
   invoke(next)
  else
   establishes fe==old(fe)
 }
 ensures  fe != null
}
```

```
class Enforce {
 …
 Fig enforce(Changed next){
  if(!next.fe.isFixed)
   invoke(next)
  else
    return fe;
 }
 when Changed do enforce;
}
```

1. TC specifies control effects independent of the implementation of the handlers Enforce, Logging, etc.
2. invoke(next) in TC assures invoke(rest) 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 outcomes.

Exercise: TC-Augmentation

- Change to directory TC-Augmentation
  - Open file Changed.java
  - Notice embedded form of contracts
  - See how handler in Update.java refines

Exercise: TC-Narrowing

- Change to directory TC-Narrowing
  - Open file Changed.java
  - Notice embedded form of contracts
  - See how contract in Enforce.java refines

Conclusion

- Motivation: intellectual control on complexity essential
  - Implicit invocation (II) and aspect-orientation (AO) help
  - ... but have limitations
- Ptolemy: combine best ideas of II and AO
  - Quantified, typed events + arbitrary expressions as explicit events
  - Translucid contracts
- Benefits over implicit invocation
  - decouples observers from subjects
  - ability to replace events powerful
- Benefits over aspect-based models
  - preserves encapsulation of code that signals events
  - uniform and regular access to event context
  - robust quantification
- Last but not least, more modular reasoning

Opportunities to Contribute

- Language design efforts
  - **Ptolemy# to come out in June, testing underway (Extension of C#)**
  - Transition to less front-end changes (for PtolemyJ)
- Verification efforts
  - More expressive support for embedded contracts
  - Practical reasoning approaches for heap effects
  - Better verification error reporting

Opportunities to Contribute

- Case study efforts – compiler supports metrics
  - Showcase applications, examples for Ptolemy
  - Comparison with other languages/approaches
- Infrastructure efforts
  - Support in Eclipse, other IDEs
  - Better error reporting, recovery
- Language manuals, descriptions,…

All are welcome!!!

Open source MPL 1.1 License

Modularizing Crosscutting Concerns with Ptolemy

Hridesh Rajan

with Gary T. Leavens, Sean Mooney, Robert Dyer, and Mehdi Bagherzadeh

Twitter: @ptolemyj

Sourceforge Project: ptolemyj

This work is supported by the US National Science Foundation (NSF)
under grants CCF-10-17334 and 10-17262.