

# Mining Software Repositories for Evaluating Software Engineering Properties of Language Designs (a position paper)

Hridesh Rajan

Computer Science, Iowa State University

This work was supported in part by the US National Science Foundation under grants CNS-06-27354 and CNS-08-09217.

# Improved Separation of Concern Mechanisms

- ▶ 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.], II [Garlan, Notkin, Sullivan et al.], Hyperslices [Ossher and Tarr], AO [Kiczales et al.], etc
- ▶ Shows that there is a real need
- ▶ How do we study their effectiveness?
- ▶ Focus: Implicit Invocation (II), aspects (AO), Ptolemy

# Problem

- ▶ Researcher: How can I study the features of my language?
- ▶ Approach: Study language in the context of large systems.
- ▶ Task: Let us find some large systems with real issues.
- ▶ Manager: Max. value from a development process
  - ▶ Optimize developer productivity
  - ▶ Reduce risks in the process
- ▶ Difficult to reach research-manager consensus

- ▶ Context of the study – Ptolemy v. AO v. II
- ▶ Ptolemy: combines best ideas of II and AO
  - ▶ Quantified, typed events
  - ▶ Arbitrary expressions as events
- ▶ Several benefits over both II and AO

# Running Example: Parts of a Drawing Editor

- ▶ Elements of drawing
  - ▶ Points, Lines, etc
  - ▶ All such elements are of type `FElement`
- ▶ **DisplayUpdate:** Modularize display update policy
  - ▶ Whenever an element of drawing changes —
  - ▶ Update the display
- ▶ **MoveCheck:** Impose application-wide restriction
  - ▶ No element may move up by more than 100

# Ptolemy's Design

- ▶ Design inspired from several languages
  - ▶ II languages such as Rapide [Luckham]
  - ▶ AO languages such as AspectJ [Kiczales et al]
  - ▶ incorporates ideas from Eos [Rajan and Sullivan]
  - ▶ and from Caesar [Mezini and Ostermann]
- ▶ Like Eos, no special syntax for “aspects” or “advice”
- ▶ Novel features: event model and type system

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

# Ptolemy: Event Types

```
1 FElement evtype FEChange{  
2   FElement changedFE;  
3 }
```

- ▶ Event type: return type, name, and context variables
- ▶ Context variable declarations: names and types

# Ptolemy: Event Expressions

```
1  FElement setX(Number x) {  
2    FElement changedFE = this;  
3    event FEChange{  
4      this.x = x; this  
5    }  
6  }
```

- ▶ Event expressions name event types
  - ▶ “Line 3–5 declare an event of type FEChange”
- ▶ Context variables must be bound in lexical scope
  - ▶ Absence of binding in lexical scope is a compile-time error

# Ptolemy: More about Event Expressions

```
1  FElement setX(Number x) {  
2    FElement changedFE = this;  
3    event FEChange{  
4      this.x = x; this  
5    }  
6  }
```

- ▶ Event expressions are explicit, helps in understanding
- ▶ No handlers except at explicit event expressions
  - ▶ e.g. Line 2

# Ptolemy: More about Event Expressions

```
1  FElement setX(Number x) {  
2    FElement changedFE = this;  
3    event FEChange{  
4      this.x = x; this  
5    }  
6  }
```

- ▶ Flexible event announcement
  - ▶ Arbitrary expressions can be enclosed in event expressions
- ▶ Declarative event announcement allows optimizations
  - ▶ e.g. when no one is interested in FEChange
  - ▶ ... Lines 3–5  $\equiv$  **this.x=x; this**

# Ptolemy: Bindings

```
1 class Update extends Object{  
2   ...  
3   when FEChange do update  
4 }
```

- ▶ Declarative specification of event - handler connection
  - ▶ Similar to Eos's bindings, but uses event types to quantify
- ▶ Names an event type: to select all events of that type
  - ▶ e.g. `when FEChange ...`  $\equiv$   
 $\forall \text{event } FEChange\{\dots\} \in \text{Point, Line, etc}$
- ▶ `Update` remains name-independent of `Point`, `Line`, etc

# Ptolemy: Handlers

```
1  class Update extends Object { /* ... */
2    FElement last;
3    FElement update(thunk FElement next,
4                    FElement changedFE) {
5        FElement res = invoke(next);
6        this.last = changedFE;
7        Display.update(); res
8    }
9    ...
10 }
```

- ▶ An OO method: event closure as first argument
- ▶ Can name context variables in event type as arguments
- ▶ `invoke` expression evaluates the closure

# Ptolemy: Register Expressions

```
1 class Update extends Object{ /* ... */  
2   Update init(){  
3     register(this)  
4   }  
5 }
```

- ▶ Activates the event - handler connection made by bindings
- ▶ Specifies the receiver object for the handlers

# Main Evaluation Ideas

- ▶ Rich Version Control History Available
  - ▶ Others have utilized: a full workshop (MSR)
- ▶ Version history contains real changes
- ▶ Recent work: extracting these changes
  - ▶ Changes in terms of program semantics

# How can we Utilize Rich Version Control History?

- ▶ Useful for language designs claiming better modularity
- ▶ Parnas's Information Hiding Modularity Notion
- ▶ Ability to withstand change characterizes modularity

# Evaluation Method

- ▶ Select an initial version for candidate projects
- ▶ Use Concern Mining Techniques to Semi-automatically Identify Fragmented and Scattered Concerns for Modularization
- ▶ Use Ptolemy and Alternative II and AO Techniques to Modularize Identified Concerns
- ▶ Replay Changes on All Versions Using Version History
- ▶ Software Evolution Analysis
- ▶ Existing Metrics could also be used

## Key (and difficult) Questions

- ▶ How do you replay changes?
- ▶ OO languages, reasonably understood [Dig *et al* 2007]
- ▶ Initial ideas: semantics for transformation between languages
- ▶ A number of ideas in this area [Lerner and Chambers]
- ▶ Responsibility: language designer - write transformation from base language to new language