ENHANCING BASE-CODE PROTECTION IN ASPECT-ORIENTED PROGRAMS

Mohamed ElBendary and John Boyland
University of Wisconsin-Milwaukee
Outline

- Introduction - Motivation
- Our AOP Modularity Focus
- Interface Image (I2) Approach
- I2 implementation
- I2 Evaluation
- Related Work
- Conclusion
Introduction - Motivation

- Separation of crosscutting concerns
- Roadblocks to AOP adoption
  - Not just education
  - Reality of coding standards for small companies
  - Lack of invasiveness regulation
  - Pure obliviousness
- Support for AOP adoption has to come at the language level.
Introduction - Motivation

- Interfaces Role Overlap:
  - Base code sees: Service Access Points
  - Aspect code sees: Event Hooks

- Protection (invasiveness control) is easier when roles are separated
Our AOP Modularity Focus

- Independent evolution of components
- Expanding parallel development
- Enhancing module protection
- Supporting modular reasoning
Classical AOP Limitations

- In our context, Classical AOP means: Pure Obliviousness
- Tight coupling between aspects and base code
- Base code cannot regulate any advising activity on itself
- Impossible to reason about a base code component solely by examining its interface (Tool support can help with this)
Interface Image (I2) Approach

- What is an Interface Image?
- “image” construct syntax
- “image” construct semantics
- What does I2 offer?
What is an interface image?

- A language mechanism for exporting views of a component’s advisable interface
- A middleware through which all advising is carried out
- A language mechanism for base code to express advising constraints
The “image” construct

```plaintext
image {
    [opento: {aspects allowed ITD’s} ]
    [alias definitions]
}

- An empty image scope reduces I2 to AspectJ-style AOP
```
[modifiers] RT method-name(P) =
[modifiers] RT alias(P) { Constraints }
modifiers: Java-style method modifiers
RT: return type
method-name, alias: Java-style method identifier
P: Java-style method parameter list
Constraints: A list of advising constraints
Constraints: kind clause

- Kind: \{Advice\_Kind*\}
- Advice\_Kind: before | after | after\_returning | after\_throwing | around
Constraints: (origin, boundary)

- (origin=ORIGIN, boundary=BOUNDARY);
- ORIGIN: internal | external
- BOUNDARY: method | class | package
Constraints: exceptions clause

- Exceptions: \{Exception_Type*\}
- Exception_Type: Java-style type identifier
“image” Construct Semantics

- Only classes declaring images are advisable
- Omitting a clause implies no constraint
- Empty “kind” list implies no advice allowed
- Empty “exceptions” list implies no checked exceptions can be softened
“image” Construct Semantics

- “opento” semantics
- “kind” semantics
- “(origin, boundary)” semantics
- “exceptions” semantics
Alias Definition Rules

- A class can only alias methods it declares
- Multiple (distinct) aliases for the same aliased method allowed
- Alias definitions in a base class are advisable in derived class unless method private in base
Example: Point class

Class Point extends Shape {
    protected int x, y;
    public void moveby(int dx, int dy) {
        x += dx; y += dy;
    }
    // image goes here (next slide)
}
Example: Point class

Image {
    openTo: {CheckScence};
    public void moveBy(int dx, int dy) =
    public void translate(int dx, int dy) {
        kind: {after};
        (origin=external, boundary=class);
        exceptions: {SceneInvariantViolation};
    }
}
Example: Rectangle class

class Rectangle extends Shape {
    void moveby(int dx, int dy) {
        p1x += dx; p1y += dy; p2x += dx; p2y += dy;
    }
    image {
        void moveby(int, int) = void translate(int, int){}
    }
}
}
Example:
CheckSceneInvariants aspect

aspect CheckSceneInvariants {
  pointcut moves(): call (void Shape+.translate(..));
  after(): moves() {
    scene.checkInvariants();
  }
}
Example: modifying moveby()

class Rectangle extends Shape {
    void moveby(int dx, int dy) {
        p1x += dx; p1y += dy; p2x += dx; p2y += dy;
    }
}

image {
    void moveby(int, int) = void translate(int, int);
}

P1.moveby(dx, dy);
P2.moveby(dx, dy);
Example: Updating Point

class Point extends Shape {
    ...
    image {
        void moveby(int, int) = void translate(int, int){
            (origin=external, boundary=class);
        }
    }
}
What does I2 offer?

- A level of indirection through which all advising requests are carried out
- Provides base code qualification of classes: advisable and unadvisable
- A mechanism for base code to expose views of joinpoints along with advising constraints
What does I2 offer?

- Control over aspect invasiveness (traded for less obliviousness)
- I2 affords better parallel development and reduces aspect brittleness
- I2 advising control does not limit AOP capabilities
I2 Implementation

- JastAdd
  - Error Checking
  - AST Rewrite
- abc
  - Compilation Sequence
I2 Implementation

- Image checking and collecting information:
  - “opento” clause
  - “kind” clause
  - “exceptions” clause
I2 Implementation

- “image” rewrite
  - Wrapper methods introduction
  - (origin, boundary) to pointcuts
  - “around” advice
- Sample translation
- Precedence ordering aspect
Sample Translation

Privileged static imageAspect {
  public void Point.translate(int dx, int dy) {
    moveby(dx, dy);
  }
}
void around(Point p):
  target(p) && !within(imageAspect) &&
  !within(Point) &&
  call(public void Point.moveby(int dx, int dy)){
    p.translate(dx, dy);
  }
}
Precedence Ordering Aspect

public aspect _internalOrderingAspect{
    declare precedence: *..*imageAspect*. *;
}
Compilation Sequence

- Image checking happens after computing intertype declarations
- Image rewrite and precedence ordering aspect
- Computing advice lists
- Filtering advice
- Weaving
Evaluation: Quantitative

- What are we measuring?
- How are we measuring it?
- Evaluation examples
- Results
What are we measuring?

- We measure coupling between aspects and base code classes
- Coupling is measured in terms of crosscutting relationships
- Crosscutting relationships result from advice and intertype declarations
How are we measuring it?

- Simulating effects of I2 syntax for AJDT
- Input to AJDT
Evaluation Examples

- Subject/Observer Protocol (1p, 6c, 2a)
- A Simple Telecom Simulation (1p, 10c, 2 a)
- Ants Simulation (11p, 33c, 11a)
Results

- I2 induces 26.3% more coupling for Subject/Observer Protocol
- I2 reduces coupling by 20% for Telecom Simulation
- I2 reduces coupling by 6.6% for Ants Simulation
Results

- Subject/Observer has only one advice, not much room for decoupling with aliases
- The use of “opento” introduces crosscutting relationships that were not existing in the original implementation
Results

- The more aspects use advice, the more the payoff (more room for aliasing)
- Ants Simulation is closer to real AOP programs in terms of the feature-mix. So it’s result is a better representative of effects of aliasing
Related Work

- Open Modules (2004)
- AAI (2005)
- XPI (2006)
- EJP (2007)
- MAO (2007)
- Key distinction
Differences from Open Modules

- Loose coupling without restricting advising
- I2 exposes an explicit set of joinpoints versus compact OM pointcuts
- Flexible joinpoint aliasing and advising constraints
Differences from AAI

- In I2, class is oblivious to which aspect will be extending its interface (except with opento)
- Improved readability
- Loose coupling between base code and aspect code
Differences from XPI

- In I2, joinpoints and constraints are the responsibility of the base code while pointcuts and advice are of the aspect code.
- In I2, all advice is channeled through images.
- Documentation of entry points into the class interface.
Differences from EJP

- EJP can advise arbitrary blocks of code, l2 cannot
- EJP requires advising markers to be placed manually in the source code, l2 does not
- EJP does not incorporate advising constraints on the base code side
Differences from MAO

- MAO supports better modular reasoning in exchange for less feature-obliviousness
- Control effects and heap effects
- I2 engages the base code while MAO engages the aspect code for protection
Ptolemy

- Solves the fragile pointcut problem using typed events that pointcuts can be written in terms of.
- \texttt{I2} still relies on aliases so pointcuts are as stable as the aliases.
- \texttt{I2} relies on the predefined possible events of AspectJ.
Key Distinction

- I2 recognize that interface specifications (e.g. method signatures) are intended to play two different roles in one breath:
  - Service Access Points
  - Joinpoints for use by aspects

- I2 reassigns these responsibilities by introducing the image construct and removes the role overlap
Conclusion

- It is possible to realize a design that loosely couples the evolution of base code interfaces from the AO code advising those components.
- It is possible to afford better parallel development and maintainability in exchange for less obliviousness.
Conclusion

- It is possible to provide a level of protection to the base code without restricting AO capabilities.
- Aid to modular reasoning in the presence of aspects.
- Achievable while maintaining a practical level that facilitates AOP adoption.
Thank You!