Supporting Dynamic Crosscutting with Partial Behavioral Reflection: a Case Study

Leonardo Rodríguez [1] Éric Tanter [2,3] Jacques Noyé [4,3]

(1) Universidad de la República, Instituto de Computación, Montevideo, Uruguay
(2) Universidad de Chile, Santiago, Chile
(3) OBASCO Project, Ecole des Mines de Nantes – INRIA, Nantes, France
(4) INRIA Rennes, France

Agenda

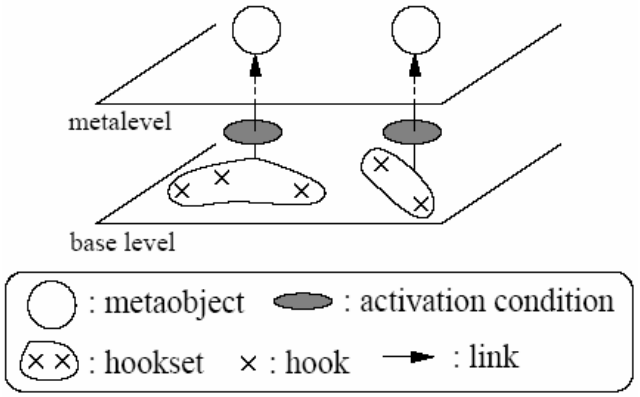
- Context and Motivation
- Naïve Mapping
- Issues and Extensions
- Mapping Revisited
- Implementation and Benchmarks
- Conclusions

Context and Motivation

- Separation of concerns
  - Reflection
  - Aspect Oriented Programming
    - Various emerging AO Languages
      - Domain specific / General propose
- Strong connection between both approaches
- Problems with the various AO Languages
  - “Reinvent the wheel” each time
  - Lack of compatibility
- We claim that an AO Kernel is needed
  - Simplify new AO language development
  - Support for composition and collaboration
- Partial Behavioral Reflection (Reflex) is our proposal
  - Validation is needed

Partial Behavioral Reflection

- Reflection
  - Needed only at well known parts of the program
  - Jumps to the meta level are expensive
- Partial Behavioral Reflection relies on high selectivity
- Basic model
  - Metaobjects (MO) acting upon reification described in terms of operations
  - Hookset gather execution points scattered in various objects
    - Hooks are base-level pieces of code that perform the reification
  - Links bind a hookset to a MO
    - Explicit and highly configurable (scope, control, activation, etc)



Partial Behavioral Reflection

- Reflex is an open implementation of this model for Java
  - No fixed Meta Object Protocol (MOP)
    - Provide a standard MOP
  - Hookset
    - Primitive
      - Consists of an operation, class selector and operation selectors
    - Composite
      - Made of other Hooksets

AspectJ

- Extends the Java language with a new unit of crosscutting concern modularity, Aspects
- Support for
  - dynamic crosscutting
  - static crosscutting
- One new concept “Join Point”
  - Kind (method-call, field-set, etc)
  - Join point shadow
  - Join point residue
- Follow the Advice and Pointcut Model
  - Pointcuts are a mean to
    - group join points of interest
    - specify context information to be passed to the aspects
  - Advice define a behavior to apply upon join points occurrences

AspectJ

- Pointcut are defined in terms of
  - pointcut designators (PCD)
  - logical operators to combine them
  - e.g.

```
pointcut move(int x,int y):call(* Point.moveXY(..)) && args(x,y)
```

- Advices are bound to a pointcut
  - Kind (before, after, around, etc)
  - The proceed statement allow to invoke the original computation replaced by an around advice
  - e.g.

```
void around(int x,int y): move(x,y){
  proceed(max(0,x), max(0,y));
}
```

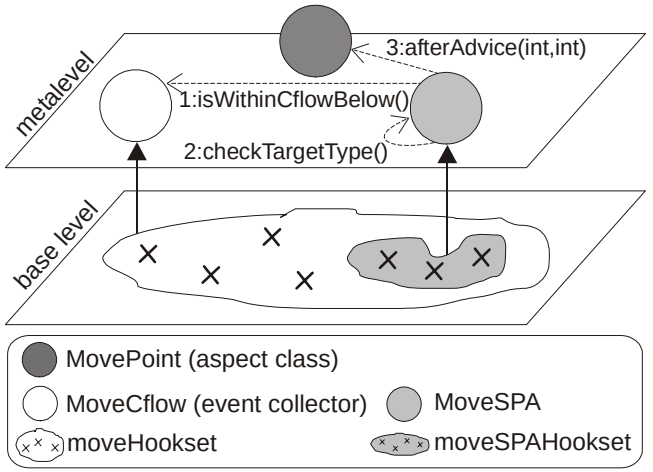
Naïve Mapping

- Mapping scope
  - Focus on dynamic crosscutting main features
  - Leaving out aspect instantiation and composition
- Pointcut mapping
  - Statically matched pointcuts corresponds to hooksets
    - e.g. `pointcut move (): call (* *.moveXY(int, int))`
  - Dynamically matched pointcuts corresponds to hookset + dynamically evaluated condition at the MO
    - e.g. `pointcut movePoint(): move() && target (Point)`
  - CFlow relations between join points are modeled with “Event Collectors”
    - e.g. `pointcut moveSinglePoint(): movePoint() && !cflowbelow(move())`
  - Context exposition is done by filtering the reified information at the MO
    - e.g. `pointcut moveSinglePointArgs(int x, int y): moveSinglePoint() && args(x,y)`

Naïve Mapping

- Aspect and Advice mapping
  - An aspect is mapped to an ordinary class
  - An advice is mapped to a method
    - The binding between the hookset (of the pointcut) and the metalevel is carry on by a link
      - Its control is set according to the advice kind
- Example

```
aspect MovingPoint {
  ...
  after(int x,int y):
       moveSinglePointArg(x,y){
    log.print("Point moved:"
              + x + "," + y);
  }
}
```



Issues and Extensions

- The naïve mapping reveal two issues
  - Residues must be checked at the metalevel
  - All context information must be reified
- MOP descriptors (first extension)
  - Describes how an operation should be reified
    - expected type, method, parameters
  - Can be specified at hookset and/or link level
  - Define custom parameters

```
new Parameter(){
  public String getCode(Operation aOp){
    return "Thread.currentThread()";
  }}
```

Issues and Extensions

- Hookset restrictions (second extension)
  - Dynamically evaluated conditions
    - fixed in the hookset generated code
  - Restriction logic is specified in static methods
  - Can be specified at hookset and/or link level
  - e.g.

```
pointcut movePoint():
  call (* *.moveXY(int, int)) && target(Point)

public static boolean accept(Object o){
  return o instanceof Point;
}
```

Mapping Revisited

- Benefits of the extensions
  - Hookset restrictions check dynamically evaluated conditions (residues)
  - MOP Descriptors specify the information to expose and the method to invoke
- Overall mapping
  - An aspect corresponds to a MO
  - An advice corresponds to a method in the MO
  - A pointcut is a Set of (Hookset, Restriction, MOP Descriptor)
    - Link binding the composite hookset with the Aspect MO
    - Additional links and event collector MO are needed for cflow restrictions

Mapping Revisited

- Pointcut translation process overview
  - Build a tree isomorphic to the AST (pointcut definition)
  - PCD are replaced by quadruples (P, S, D, C)
    - P represents an operation (kind restriction)
    - S is an expression that represents a statically matched restriction (e.g. location restrictions)
    - D is an expression that represents a dynamically matched restriction (e.g. cflow residue)
    - C is a list of context exposed information
  - Reduce the tree by eliminating all the && and ! operators
    - The final tree must have one leaf or a || operators in all its nodes

Mapping Revisited

- The *proceed* statement allows to invoke the original computation replaced by an around advice
  - Reflex supports dynamic operations which allow to invoke a reified operation
  - The runtime information of the operation occurrence is needed
- Around advices that call proceed are configured to receive an extra parameter, a command object wrapping the dynamic operation
  - It must replace the proceed parameters before invoking the dynamic operation
  - Build the appropriate operation and execute it

Mapping Revisited

- AspectJ provide join point reflective information via implicit variables
  - thisJoinPoint, thisJoinPointStaticPart, thisEnclosingJoinPointStaticPart
- Providing this variables implies
  - a set of classes implementing the interfaces of the AspectJ API
  - three MOP Descriptor parameters to instantiate those classes with the necessary information
  - advices will receive an extra parameter for each variable used

Implementation and Benchmarks

- Implemented using AspectJ Compiler and Reflex
  - offline translation
- Benchmarks

  ajc 1.2

Features	*Scenario*	*AJC*	*AJP*	Δ
before	**(w/o context)**			
no residue		1542	1705	10%
instanceOf	match	1185	1305	10%
	no match	868	894	2%
cflow	match	841	951	13%
	no match	998	1218	22%
before	**(w/ context)**			
no residue		4533	4616	1%
instanceOf	match	3241	3034	-6%
	no match	884	904	2%
cflow	match	10295	11102	7%
	no match	697	647	-7%
around	**(w/ context)**			
around	always proceed	3404	5721	68%
	half proceed	4416	6349	43%
	never proceed	5711	6990	22%

Conclusions

- PBR is expressive enough to support AspectJ dynamic crosscutting
  - The extensions provide a more natural and direct mapping
- Benchmarks shows very good results comparing to AspectJ Compiler
- First validation of Reflex as an AOP Kernel

Future work

- Support
  - Full support for dynamic crosscutting
  - Support for static crosscutting
- Optimize implementation
  - Better benchmarks results (e.g. proceed)
- Deeper benchmarks