Arranging language features for more robust pattern-based crosscuts

Kris Gybels and Johan Brichau

kris.gybels@vub.ac.be
johan.brichau@vub.ac.be

Vrije Universiteit Brussel
Outline

• Problems caused by limited crosscut languages
• Features for more flexible languages
• Problem example solved
• Optimization opportunities for weavers
- New methods are not automatically captured by aspect
- Pointcut is simply an *enumeration* of wanted methods
- Pattern-based crosscut
- But *pattern* is based on naming convention *arranged* for in Buffer
Structuring classes for the sake of an aspect = Arranged Patterns

- Naming conventions
- Method annotations
- Calling a special dummy method
- Putting code in a specific package
- Refactoring the code to expose joinpoints
- ...

Exploiting program structure in aspects not always problematic
Problematic when relationship between class and aspect reverses
Another look at what happens

Why?

- Programmers too lazy?
- Crosscut language not powerful enough?
- ...

BufferObserver

pointcut changesState():

  execution( * Buffer.put*(*) )
  || execution( * Buffer.get*(*) )

The real semantics of the pointcut, “when state changes”

Large discrepancy between the two descriptions of the pointcut
Observer

Rule changesState(?class, ?methodName) if shadowIn(?class, ?methodName, ?sp), assignmentShadow(?sp, ?variable)


Crosscut language based on logic programming and reification of program structure

Language features:

- Unification
- Reasoning about joinpoint properties
- Link to shadow joinpoints
- Reusable parameterized rules
- Recursion
Joinpoint, static & dynamic structure predicates

Send

reception

reference

assignment

blockExecution

& inObject, class, ...

Joinpoints similar to AspectJ: key events in execution of OO program (Smalltalk)
Expressing advices

**Observer**

```
after ?jp matching
  reception(?jp, ?msg, ?args),
  inObject(?jp, ?obj),
  objectClass(?obj, ?class),
  changesState(?class, ?msg),
  not(caller(?jp, ?obj))
do
  observers notify
```

Crosscut as logic query over joinpoints

Advice action as regular Smalltalk code
Basis of pattern matching in logic programming

?jp matching
reception(?jp, ?methodName, <?firstArgument, 5>)

add: 8 to: 5

?methodName U add:to:
<?firstArgument, 5> U <8,5>

add: 8 to: 9

?methodName U add:to:
<?firstArgument, 5> U <8,9>
Reasoning about properties

- More complex conditions on joinpoints
- Not just ‘should have this value’

?jp matching
    reception(?jp, withdraw:, <?amount>),
    inObject(?jp, ?obj),
    objectVariable(?obj, balance, ?balance),
    difference(?balance, ?amount, ?afterWithdrawal),
    below(?afterWithdrawal, 0)
Buffer

put: anObject

size := size + 1

size := size + 1

size := size + 1

joinpoints, capture
with send(?jp, ..., ...)

shadow point, capture
with shadowOf(?sp, ?jp)

- Links dynamic (joinpoint) view of program to static view
- Allows reasoning about static structure of code
- Parameterized reusability mechanism of logic programming
- Used to make reusable crosscuts

Rules vs. procedures/... etc:
- in/out parameters
- multiple implementations
- multiple solutions

Allow recursion or not?
- Useful for recursive patterns
<table>
<thead>
<tr>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule changesState(?class, ?methodName) if shadowIn(?class, ?methodName, ?sp), assignmentShadow(?sp, ?variable)</td>
</tr>
<tr>
<td>Method (possibly) changes state if it does an assignment</td>
</tr>
<tr>
<td>Rule changesState(?class, ?methodName) if shadowIn(?class, ?methodName, ?sp), messageShadow(?sp, ?rcvr, ?msg), selfReceiver(?rcvr), changesState(?class, ?msg)</td>
</tr>
<tr>
<td>Method (possibly) changes state if it calls another method that is state changing</td>
</tr>
<tr>
<td>after ?jp matching</td>
</tr>
<tr>
<td>reception(?jp, ?msg, ?args), inObject(?jp, ?obj), objectClass(?obj, ?class), changesState(?class, ?msg), not(caller(?jp, ?obj))</td>
</tr>
<tr>
<td>When state changing message received that was not sent by the object itself</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>observers notify</td>
</tr>
</tbody>
</table>
Weaver efficiency?

Crosscuts can depend on runtime data

Naive weaver implementation:
“at every joinpoint check all crosscuts for matches”

Optimization opportunity:
crosscuts also state conditions on static data

?jp matching
  reception(?jp, test:, ?argument),
  below(?argument, 10)

Optimized weaver:
- simplistic “partial evaluation”
- evaluate crosscuts at compile time
  using partial information and ternary logic
Summary

- Potential pitfalls of crosscuts
  - Not able to deal with change
  - Able to deal with change but through arranged pattern
- Needed: better crosscut languages
- Our approach: logic + joinpoint, program structure predicates (LMP)
  - Reasoning about joinpoint properties
  - Reasoning about shadow points
  - Recursive patterns
  - Expressive crosscut language
Conclusion: “Lessons learned for …”

- Aspect programmers
  - Avoid “arranged pattern” trap
- Aspect/crosscut language designers
  - Assist programmers with more flexible languages
- Weaver implementers
  - Use advanced language interpretation techniques