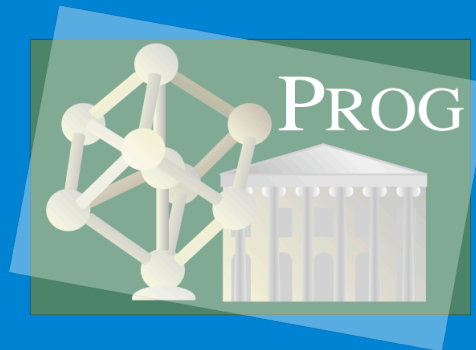


Arranging language features for more robust pattern-based crosscuts

Kris Gybels and Johan Brichau

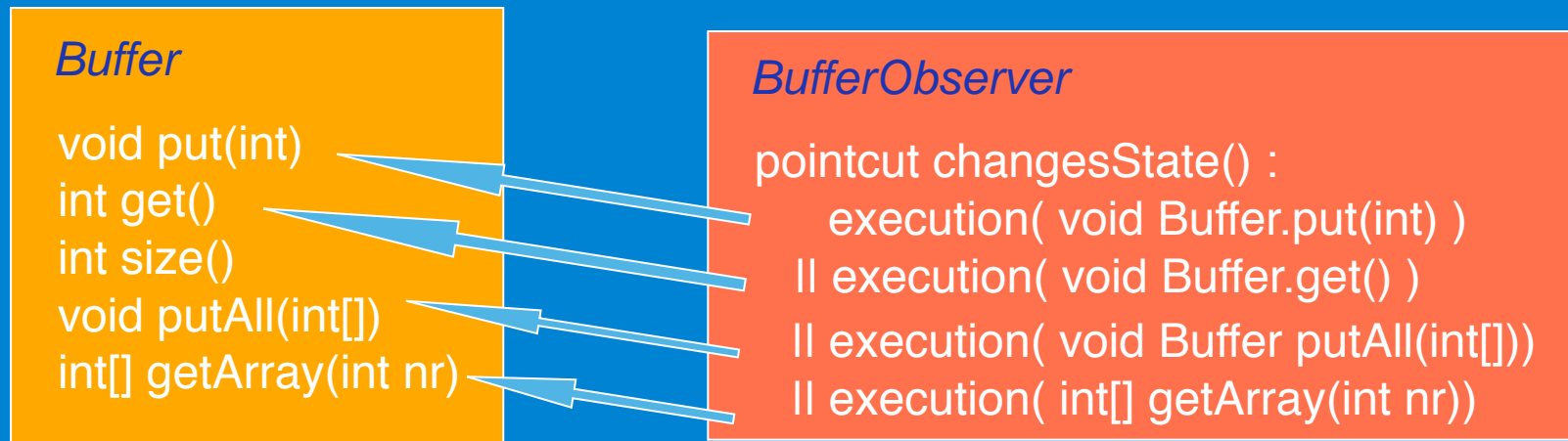
kris.gybels@vub.ac.be
johan.brichau@vub.ac.be



Vrije Universiteit Brussel

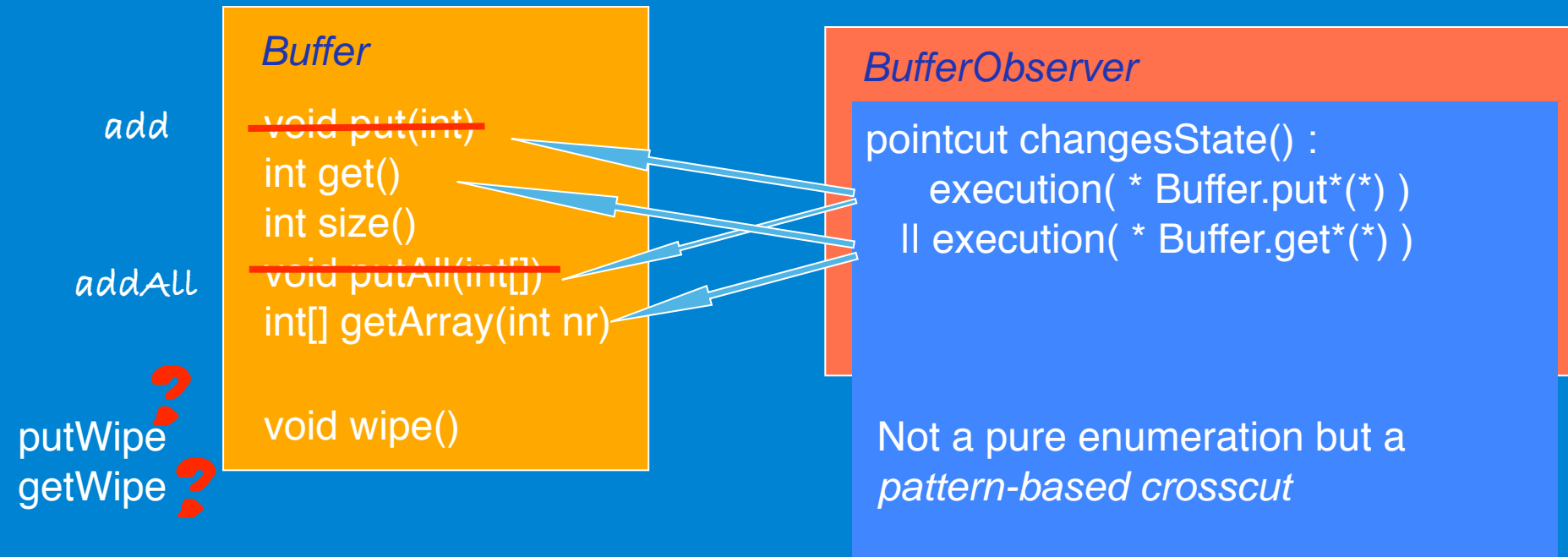
- Problems caused by limited crosscut languages
- Features for more flexible languages
- Problem example solved
- Optimization opportunities for weavers

A ~~simple AOP~~ example problem



- New methods are not automatically captured by aspect
- Pointcut is simply an *enumeration* of wanted methods

A better version?



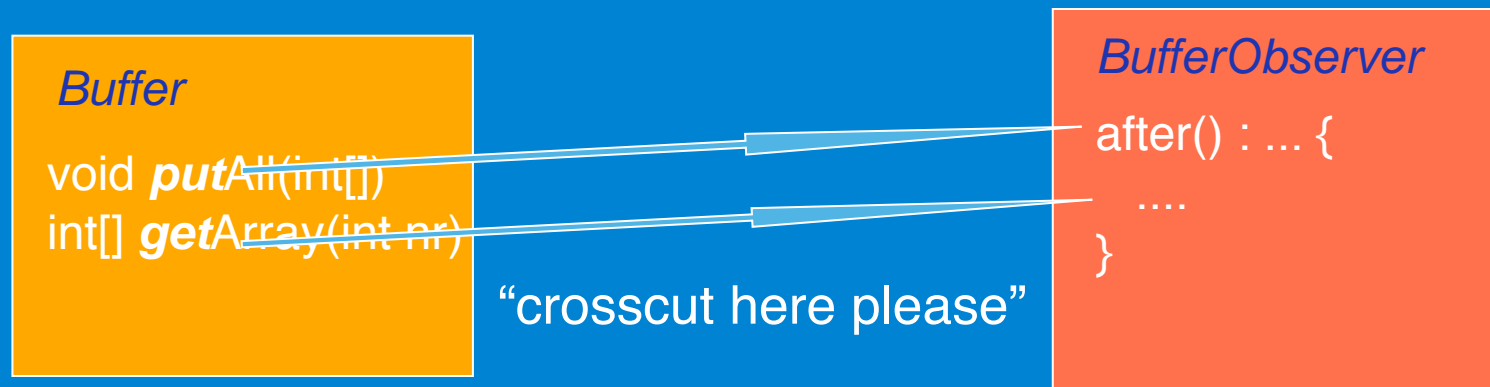
- Pattern-based crosscut
- But *pattern* is based on naming convention *arranged* for in *Buffer*

Arranged Patterns

Programming Conventions ~~≠~~ Arranged Patterns

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 semantics as communicated to the aspect, “all methods whose name begins with put or get”

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

Large discrepancy between the two descriptions of the pointcut

An experiment

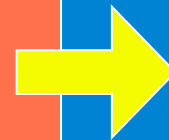
Observer

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

```
Rule changesState(?class, ?methodName) if  
  shadowIn(?class, ?methodName, ?sp),  
  messageShadow(?sp, ?obj, ?msg, ?args),  
  selfBe(?sp, ?obj)
```

**Crosscut language based on
logic programming and
reification of program structure**

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



Language features:

- Unification
- Reasoning about joinpoint properties
- Link to shadow joinpoints
- Reusable parameterized rules
- Recursion

The Language

send(?jp, ?selector, ?arguments)

reception(?jp, ?selector, ?arguments)

reference(?jp, ?varName, ?value)

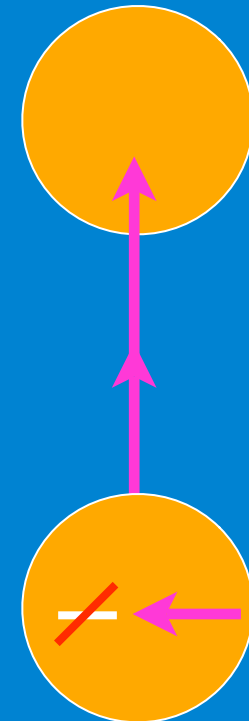
assignment(?jp, ?varName, ?value)

blockExecution(?jp, ?args)

& inObject, class, ...

Joinpoint, static & dynamic structure predicates

Logic Programming



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
```

Advice action as regular
Smalltalk code

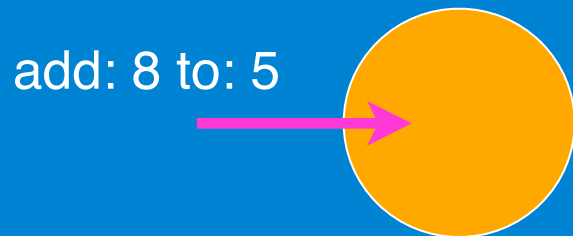
Crosscut as logic query
over joinpoints

Unification

Basis of pattern matching in logic programming

?jp matching

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



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



?methodName U add:to:
<?firstArgument, 5> ~~U~~ <8,9>

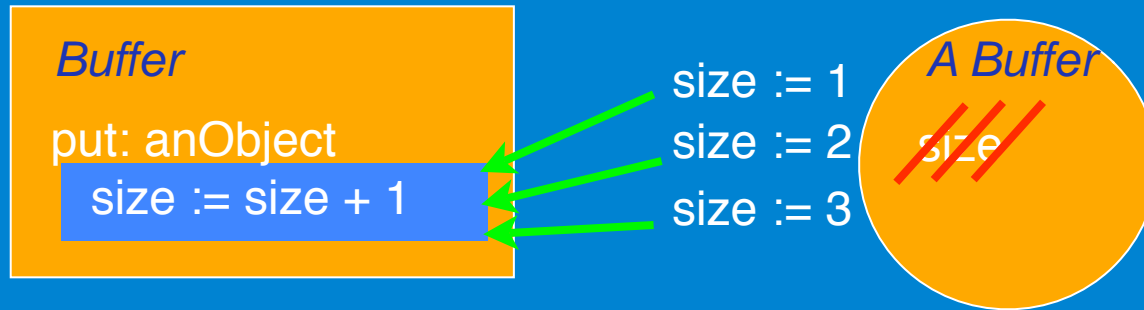
Reasoning about properties

- 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)
```

Link to shadows



shadow point, capture with shadowOf(?sp, ?jp) ← joinpoints, capture with send(?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

Observer

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

Method (possibly) changes
state if it does an assignment

Rule changesState(?class, ?methodName) if
shadowIn(?class, ?methodName, ?sp),
messageShadow(?sp, ?rcvr, ?msg),
selfReceiver(?rcvr),
changesState(?class, ?msg)

Method (possibly) changes
state if it calls another method
that is state changing

after ?jp matching

reception(?jp, ?msg, ?args),
inObject(?jp, ?obj),
objectClass(?obj, ?class),
changesState(?class, ?msg),
not(caller(?jp, ?obj))

When state changing
message received that was
not sent by the object itself

do

observers notify

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

- 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