Concerning Efficient Reasoning in Aspect-Oriented Languages

Gary T. Leavens
Iowa State University
Department of Computer Science
Based on work with Curtis C. Clifton and James Noble

July 30, 2007

Concerning Efficient Reasoning in Aspect-Oriented Languages

Gary T. Leavens
University of Central Florida
School of Electrical Engin. and Computer Science
Based on work with Curtis C. Clifton and James Noble

July 30, 2007

Summary

Problem:

- Efficient reasoning
  in Aspect-Oriented languages

Approach:

- Use static analysis,
  identify (non-)interference

Background: Reasoning

- Specification, of:
  - Object state
  - Method:
    - Preconditions
    - Heap effects (postcondition + frame)
    - Control effects
- Verification, of method:
  - Calls
  - Implementation

Tally Specification

```
public class Tally {
  protected /*@ spec_public @*/ int val = 0;

  /*@ requires true;
    @ assignable this.val;
    @ ensures this.val == \old(this.val + inc);
    @*/
  public void add(int inc) { this.val += inc; }
}
```

Call Verification: Heap Effects

```
public void testAdd(Tally t) {
  //@ assert t.val == 0;
  t.add(-10);
  //@ assert t.val == -10;
}
```

Implementation Verification: Heap Effects

```
public void add(int inc) {
  //@ assert true;
  this.val += inc;
  //@ assert this.val == \old(this.val + inc);
}
```

Implementation Verification: Heap Effects

For all normal states, *pre*,

if $\mathcal{E}$[[t.val == 0]](*pre*)

then let *post* = $\mathcal{S}$[[t.add(-10)]](*pre*)

in if normal(*post*)

then $\mathcal{E}$[[t.val == -10]](*post*)

else *true*

else *true*

Implementation Verification: Frame Axiom

//@ **assignable this**.val;

- Conservative static analysis, accumulates:
  - Assignments
  - Assignable clauses for calls

Call Verification: Frame Axioms

```
public void testAddFrame(Tally t, C c, H h) {
  //@ assert t != c;
  //@ assert t.val == 0 && c.val == 7 && h.b;
  t.add(-10);
  //@ assert t.val == -10 && c.val == 7 && h.b;
}
```

Costs of Reasoning

- Specification effort
- Verification effort for calls
  - Find specification
  - Prove precondition
  - Show frame independent of preserved part
  - Show postcondition implies assertion

Benefits of Reasoning with Contracts

Maintainable despite changes to:

- Implementation
- Subtypes

Modular:

- Only look at small part of program
- Gives scalability

Background: AspectJ

- Features:
  - Law enforcement (declare error/warning)
  - Intertype declarations (adding fields/methods)
  - *Advice on dynamic execution events*

Background: Advice in AspectJ

- *Join point* = potential dynamic event
  - Call of method / constructor
  - Execution of method / constructor body
  - Get / set of field
- Before advice – run before join point
- After advice – run after join point
- Around advice – run instead of join point

Example: Counting Calls

```
public aspect C {
  private /*@ spec_public @*/ int val = 0;

  public pointcut tallyAddCalls() :
                        call(* Tally+.add(..));

   before() : tallyAddCalls() { this.val++; }
}
```

Problem: Frame Axiom Invalid?

```
public void testAddFrame(Tally t, C c, H h) {
  //@ assert t != c;
  //@ assert t.val == 0 && c.val == 7 && h.b;
  t.add(-10);
  //@ assert t.val == -10 && c.val == 7 && h.b;
}
```

Problem Analysis

With before / after advice:

- Calls do more
  - Before advice
  - Call
  - After advice
- Specification doesn't reflect that
- Verification not designed for it

Example: Buffering Calls

```
public aspect BufferTally {
  private int tallies = 0;
  void around(int i) :
                    call(* Tally+.add(..)) && args(i)
  {
    this.tallies += i;
    if (i == 0 || Math.abs(this.tallies) > 100) {
      proceed(this.tallies);
      this.tallies = 0;
} } }
```

Call Verification: Control Effects

```
public void testAdd(Tally t) {
  //@ assert t.val == 0;
  t.add(-10);
  //@ assert t.val == -10;
}
```

Problem Analysis

With advice:

- Control effects:
  - Replacing call
  - Running it multiple times
  - Not returning (exception, abort)
- Specification doesn't reflect that
- Verification not designed for it

Problem Summary

- How to reason efficiently?
  - How much of program?
  - What changes can be ignored?
  - Which changes need how much effort?

Approach -1: Use Semantics Directly

Specification = code

Verification:

- Find applicable advice (Eclipse AJDT)
- Weave (recursively)
- Use semantics

Approach -1: Use Semantics Directly

Benefits:

- Maximally expressive
- Doesn't restrict programmers

Costs:

- All applicable changes need re-verification
- No abstraction

Approach 0: Functional Advice

- Advice with no heap or control effects

Benefits:
- Base code reasoning unaffected

Costs:
- Useless

Approach 1: “Harmless” Advice Dantas and Walker (POPL 2006)

- No information flow from advice to base
- Conservative static analysis
- Base code assertions
  can’t mention advice state

Approach 1: "Harmless" Advice
Dantas and Walker (POPL 2006)

Benefits:

- No annotations needed
- No heap effects on base

Costs:

- No help with control effects
- Loss of expressiveness
  - Some aspects (assertions) can't be written
- No help with interference among advice

Approach 2: Behavioral Subtyping

OO Analogy:

- Around advice ~ overriding method
- Proceed ~ super call

Behavioral Subtyping:

- Advice obeys specification of all it advises

Approach 2: Behavioral Subtyping

Benefits:

- Verification of base code independent of advice

Costs:

- Quantification limits in practice
- Re-verify advice when advise more
- Much advice outside this paradigm (e.g., Buffering)

Approach 3: Limits on Advice

- Gudmundson and Kiczales (2001)
- Griswold *et al.*'s XPIs (2005-6)
- Aldrich's Open Modules (2005)

Idea:

- Advice only on declared pointcuts

Approach 3: Limits on Advice

Benefits:

- Some code can't be advised
- Enables negotiation
- No limits on expressive power

Costs:

- Extra annotation / code
- No help where advice can be applied
- No help finding interference among advice

Similar Approaches

- Composition Filters: no execution advice
- HyperJ: limits quantification
- Larochelle *et al*.: hide join points
- Ossher: confirm or deny advice application
- Lopez-Herrejon, Batory: limit quantification
- Cottenier *et al*.: limit quantification
- Rajan-Leavens: no obliviousness (not AOP)

Approach 4: Weave Specifications

- Specify:
  - Object state and methods
  - Aspect state and advice
    - Heap effects
    - Control effects
- Weave specifications

Approach 4: Weave Specifications

Benefits:

- More abstract than code
- Allows changes in methods and advice

Costs:

- Lack of expressiveness?
- Weaving specifications is hard / expensive

Optimizations for Weaving Specifications

- Inapplicable advice ignored
- Spectator advice ignored:
  - Want

    Advice ∘ Call ≅ Call
  - Problem: soundness
- Other advice:
  - Advice ∘ Call ≅ *weave*(Advice, Call)
  - Problem: expense

Where the Composition is Done (Clifton 2005)

- Client utilities:
  client weaves into call semantics
- Implementation utilities:
  implementation of method weaves into its specification

Approach 4a: Optimization via Effect Analysis

- Advice A *heap interferes with* code C iff: A writes a field that C reads
- Efficiency: only look at signatures

Can apply to both:

- Advice vs. base code
- Advice vs. other advice

Effect on Specification Composition

- Want non-interference to imply:
  weave(**ensures** *P*, **ensures** *Q*)
  $\cong$ **ensures** *P && Q*
  (modulo control effects)

- For spectators, projection onto base fields can ignore advice's effects

Potential Cost: Overly Conservative Analysis

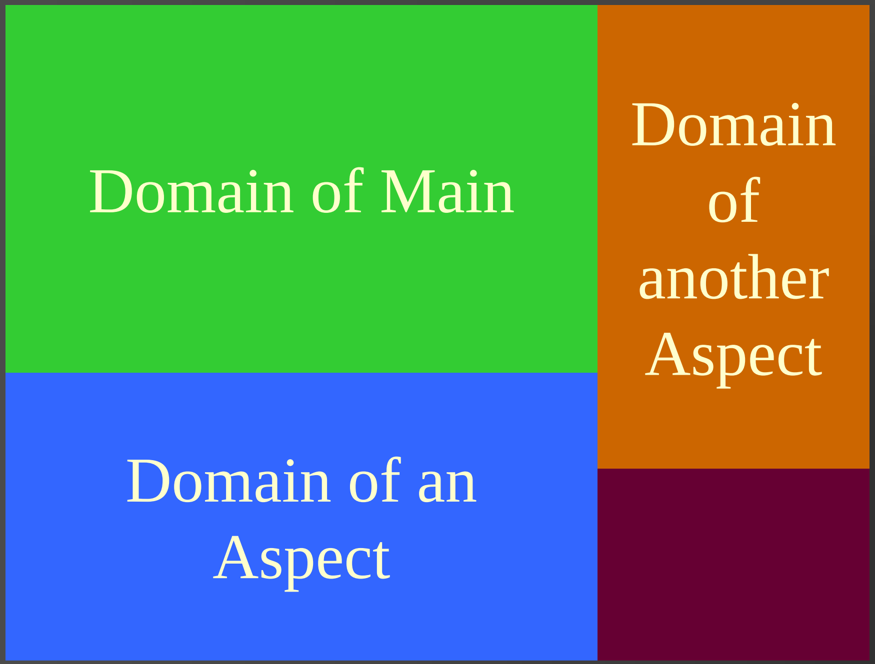
- Even spectators will have side effects

```
private int val = 0;
before() : tallyAddCalls() { this.val++; }
```

Concern Domains (Clifton 05) (Clifton, Leavens, Noble 07)

- Declare concern domains (heap partitions)
- Declare write effects (and control effects)
- Uses readonly types
- Type/effect analysis
  detects potential interference
- Sound for checking possible interference

Concern Domains Partition the Heap



Example MAO Class with Concern Domains

```
public class CDTestTally<Owner> {
  @writes({"Owner"})
  public void testAdd(Tally<Owner> t)
  {
    //@ assert t.value == 0;
    t.add(-10);
    //@ assert t.value == -10;
  }
}
```

Example MAO Aspect with Concern Domains

```
@readonlyDomains({"Other"})
@depends({ @varies({"Owner", "Other"}) })
public aspect CDBufferTally<Owner, Other>  {
  private int tallies = 0;
  @writes({“Owner”})
   void around(int i)
         : call(* Tally<Other>+.add(..)) && args(i) {
    this.tallies += i;
    if (i == 0 || Math.abs(this.tallies) > 100) {
      proceed(this.tallies);
      this.tallies = 0;
} } }
```

Checking Spectators in MAO

A spectator aspect:

- Only has surround advice:
  - Only writes its home concern domain (Owner)
  - Does not change arguments or results
  - Does not interrupt program flow:
    - No explicit exception throwing
    - Proceeds exactly once
- Control effect guarantees enforced
- Heap effect guarantees proved sound

Analysis of Concern Domains

Benefits:

- Spectators can be ignored
- Sound for detecting heap (non-)interference

Costs:

- Declaring effects of methods and advice
- Other concern domain annotations
- Restrictions on assertions

Related Work in Static Analysis

Rinard, Salcianu, Bugrara (*FSE '04*):

- Control flow analysis and global pointer + escape analysis
- More fine-grained than concern domains
- Considers interference

- But it's a whole-program analysis

Summary

Goals:

- Reasoning efficiency
- Practicality

Approaches could be combined?

  - Applicability (AJDT)
  - Declared limits (XPIs, OMs)
  - Heap partitions / effects (MAO)
  - Specification of advice, weaving specifications
  - Other static analyses + annotations

Future Work

- Implement and do case studies
- Integrate MAO's concern domains and JML's data groups [Leino98]
  - Problem: data groups can overlap
  - Benefit: less syntax, plug into other tools

Conclusions

- Around advice like overriding method
- But often used to change behavior
- So refinement isn't a complete solution
- Efficient reasoning by:
  - Limited applicability
  - Specifications of advice
  - Weaving specifications
  - Effect analysis (concern domains)