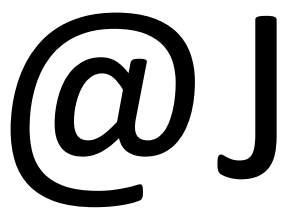
VMIL09 - October 25, 2009 - Orlando, Florida



Towards Rapid Development of Dynamic Analysis Tools for the Java Virtual Machine

Walter Binder, Alex Villazón, Danilo Ansaloni, Philippe Moret
University of Lugano, Switzerland

Background

Developing dynamic analysis tools for:

- profiling
- debugging
- testing
- reverse engineering

is difficult and error-prone

➡ time-consuming

➡ **expensive**

Aspect-Oriented Programming

With AOP you can express instrumentations at a higher abstraction level

➡ reduce developing and testing time

➡ **rapid**

Is AOP the silver bullet for dynamic analysis tools?

Overview

- AOP at a glance
- Limitations of prevailing AOP frameworks
- @J features
- Example: LiLa
- Conclusions

AOP at a glance

With AOP you can add arbitrary code *before*, *after* or *around* **any** identifiable execution point:

- method/constructor body
- method/constructor invocation
- field access
- exception handler
- ...

AOP at a glance

To count the number of object allocations:

```
aspect AllocCounter {
  final AtomicLong counter = new AtomicLong();

  after() returning(Object o) : call(*.new(..)) {
    System.out.println("New object allocated: " + o);
    counter.incrementAndGet();
  }
}
```

AspectJ terminology

Join Points are specific execution points:

- field access (read/write)
- call/execute method
- call/execute constructor
- call/execute exception handler

AspectJ terminology

Pointcuts intercept specific join points

```
aspect AllocCounter {
  final AtomicLong counter = new AtomicLong();

  after() returning(Object o) : call(*.new(..)) {
    System.out.println("New object allocated: " + o);
    counter.incrementAndGet();
  }
}
```

AspectJ terminology

Advice is the code executed before/after/around each join point intercepted by a pointcut

```
aspect AllocCounter {
  final AtomicLong counter = new AtomicLong();

  after() returning(Object o) : call(*.new(..)) {
    System.out.println("New object allocated: " + o);
    counter.incrementAndGet();
  }
}
```

AspectJ terminology

Aspects are class-like elements added to Java by AspectJ

```
aspect AllocCounter {
  final AtomicLong counter = new AtomicLong();

  after() returning(Object o) : call(*.new(..)) {
    System.out.println("New object allocated: " + o);
    counter.incrementAndGet();
  }
}
```

Limitations of AOP frameworks

AOP frameworks have not been designed for developing dynamic-analysis tools

Common limitations:

- no data passing between advice bodies
- no execution of custom code at weaving time
- no basic-block level join points

Our goal

Develop a new:

- expressive
- efficient
- portable and compatible
- comprehensive
- easy to use

aspect-oriented instrumentation framework: **@J**

Aspect Tools in Java: @J

@J is:

- an annotation-based aspect language and weaver
- based on AspectJ annotation syntax
- designed for developing dynamic-analysis tools
- compatible with standard Java compilers

@J features

New @J features:

- invocation-local variables
- snippet composition
- basic-block level join points

Invocation-local variables

Invocation-local variables are:

- accessed through public static fields
- annotated with **`@InvocationLocal`**
- mapped to local variables in woven methods

They allow data passing between snippets that are *woven in the **same** method body*

Invocation-local variables

```
public aspect TimeAspect {
    pointcut allCalls() : call(* *.*(..)) && !within(TimeAspect);

    @InvocationLocal
    public static long start;

    before() : allCalls() {
        start = System.nanoTime();
    }

    after() : allCalls() {
        long elapsed = System.nanoTime() - start;
        logExecTime(thisJoinPoint, elapsed);
    }
}
```

Snippets

Snippets are:

1. public static methods with void return type
2. annotated with:
   - **`@BeforeSnippet`**
   - **`@AfterSnippet`**
   - **`@AfterReturningSnippet`**
   - **`@AfterThrowingSnippet`**

Snippets

Snippets look similar to AspectJ advices, but:

- by default, are inlined in the woven code
- cannot be woven around a join point
- support invocation-local variables
- support additional parameters
- can be executed at weaving-time

Snippets

Weave-time executable snippets:

- are not inlined
- are executed at weaving-time
- they can only access static information
- they can change the value of invocation-local variables
- after advice-execution, the weaver inlines the code to initialize these invocation-local variables with the respective values

Basic-block join points

In @J, every basic-block of code is a join point.

@J provides:

- customizable basic-block analysis algorithm
- customizable matching properties
- customizable data properties
- clear low-level interface to BCEL

Example: LiLa

```
@J
public class LiLa {
    @InvocationLocal
    public static long start; // stores starting time of listener execution

    @InvocationLocal
    public static boolean needsProf; // stores result of static analysis

    @Pointcut( "execution(* java.util.EventListener+.*(..))" )
    void listenerExec() { }

    @BeforeSnippet( pointcut = "listenerExec"; execute = true; order = 1; )
    public static void analyzeNeedsProfiling( JoinPoint.StaticPart jpsp) {
        needsProf = isInterfaceMethod(jpsp); // not shown here
    }
...
}
```

Example: LiLa

```
@J
public class LiLa {
...
    @BeforeSnippet( pointcut = "listenerExec”; order = 2; )
    public static void takeStartTime() {
        if (needsProf) start = System.nanoTime();
    }

    @AfterSnippet( pointcut = "listenerExec && this(listener)"; )
    public static void takeEndTimeAndProfile( JoinPoint.StaticPart jpsp,
                                              java.util.EventListener listener) {
        if (needsProf) {
            long exectime = System.nanoTime() - start;
            if (exectime >= THRESHOLD_NS)
                profileEvent(jpsp, listener, exectime); // not shown here
        }
    }
...
}
```

Example: LiLa

```
class ExampleListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        doSomething();
    }

    public void notDeclaredInInterface() {
        doSomethingElse();
    }
...
}
```

Example: LiLa – woven code

```
class ExampleListener implements ActionListener {
    // representing actionPerformed
    private static final JoinPoint.StaticPart jpsp1 = ...;
...
    public void actionPerformed(ActionEvent e) {
        long start = 0L;
        boolean needsProf = true;
        if (needsProf) start = System.nanoTime();
        try {
            doSomething();
        } finally {
            if (needsProf) {
                long exectime = System.nanoTime() - start;
                if (exectime >= LiLa.THRESHOLD_NS)
                    LiLa.profileEvent(jpsp1, this, exectime);
            }
        }
    }
}
```

Example: LiLa – optimized code

```
class ExampleListener implements ActionListener {
    // representing actionPerformed
    private static final JoinPoint.StaticPart jpsp1 = ...;
...
    public void actionPerformed(ActionEvent e) {
        long start = System.nanoTime();
        try {
            doSomething();
        } finally {
            long exectime = System.nanoTime() - start;
            if (exectime >= LiLa.THRESHOLD_NS)
                LiLa.profileEvent(jpsp1, this, exectime);
        }
    }
}
```

Example: LiLa – woven code

```
class ExampleListener implements ActionListener {
    // representing notDeclaredInInterface
    private static final JoinPoint.StaticPart jpsp2 = ...;
...
    public void notDeclaredInInterface(ActionEvent e) {
        long start = 0L;
        boolean needsProf = false;
        if (needsProf) start = System.nanoTime();
        try {
            doSomethingElse();
        } finally {
            if (needsProf) {
                long exectime = System.nanoTime() - start;
                if (exectime >= LiLa.THRESHOLD_NS)
                    LiLa.profileEvent(jpsp2, this, exectime);
            }
        }
    }
}
```

Example: LiLa – woven code

```
class ExampleListener implements ActionListener {
    // representing notDeclaredInInterface
    private static final JoinPoint.StaticPart jpsp2 = ...;
...
    public void notDeclaredInInterface(ActionEvent e) {
        doSomethingElse();
    }
}
```

Ongoing research

1. Buffered snippets:
   - we already have a programming model for buffered advices integrated with AspectJ
   - support buffered snippets and their composition with inlined and weave-time executable snippets
2. New case-studies:
   - CProf, Senseo, MemoryLeak, CC Profiling, ...
3. Bytecode level join points
4. Collection of low-level metrics

Conclusion

- AOP simplifies the development of DA tools
- AspectJ lacks important features
- @J: annotation-based aspect language ad weaver
- Some new features:
  1. invocation-local variables
  2. snippet composition
  3. weave-time executable snippets
  4. basic-block level join points

Thanks!

Questions?