A Machine Model for Aspect-Oriented Programming

Michael Haupt
Software Architecture Group
Hasso Plattner Institute, University of Potsdam

Hans Schippers
Formal Techniques in Software Engineering
University of Antwerp
Agenda

- aspect-oriented programming (AOP)
  - concepts and terminology
  - weaving and weaving approaches
- motivation
  - aspect weaver targets
  - semantic gap in language core mechanism representation
- a machine model for AOP
  - virtual join points
  - model introduction by example
- future work
  - possible shape of an AOP machine
  - language implementations
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.

tangling/scattering
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.

tangling/scattering aspect
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.

tangling/scattering
aspect
join points
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.

tangling/scattering aspect
join points
pointcuts
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.

tangling/scattering aspect
join points
pointcuts
advice

Michael Haupt, Hans Schippers (michael.haupt@hpi.uni-potsdam.de, hans.schippers@ua.ac.be)
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.

tangling/scattering aspect join points pointcuts advice
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.

tangling/scattering aspect
join points
pointcuts
advice
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns. Weaving is the process of letting aspects take effect.

tangling/scattering aspect
join points
epointcuts
advice
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns.

Weaving is the process of letting aspects take effect.

tangling/scattering aspect
join points
pointcuts
advice
join point shadows
AOP Concepts and Terminology

AOP deals with the modularisation of cross-cutting concerns. Weaving is the process of letting aspects take effect.

tangling/scattering aspect join points pointcuts advice join point shadows residues
AOP Concepts and Terminology

AOP deals with the modularisation of **cross-cutting concerns**.

**Weaving** is the process of letting aspects take effect.
Weaving Approaches

- weaving **source code**
  - preprocessors implement language extensions
  - e.g., EAOP
Weaving Approaches

- weaving **source code**
  - preprocessors implement language extensions
  - e.g., EAOP
- weaving **byte code**
  - most popular approach today
  - e.g., AspectJ, ..., Steamloom
Weaving Approaches

- **weaving source code**
  - preprocessors implement language extensions
  - e.g., EAOP

- **weaving byte code**
  - most popular approach today
  - e.g., AspectJ, ..., Steamloom

- **weaving at the meta level**
  - do not touch application code, but its meta entities
  - e.g., AspectS, PROSE I (debugger)
Weaving Approaches

- weaving **source code**
  - preprocessors implement language extensions
  - e.g., EAOP
- weaving **byte code**
  - most popular approach today
  - e.g., AspectJ, ..., Steamloom
- weaving at the **meta level**
  - do not touch application code, but its meta entities
  - e.g., AspectS, PROSE I (debugger)
- weaving using **application-level entities**
  - accessible through an API, such as Java dynamic proxies
  - e.g., Spring AOP
Weaving Approaches

• weaving source code
  – preprocessors implement language extensions
  – e.g., EAOP
• weaving byte code
  – most popular approach today
  – e.g., AspectJ, ..., Steamloom
• weaving at the meta level
  – do not touch application code, but its meta entities
  – e.g., AspectS, PROSE I (debugger)
• weaving using application-level entities
  – accessible through an API, such as Java dynamic proxies
  – e.g., Spring AOP
Weaver Targets

- commonalities of weaving approaches
  - all target non-AO machines
  - map AO core mechanisms to OO mechanisms
  - AO works on target machine instead of being supported in it
Weaver Targets

- commonalities of weaving approaches
  - all target non-AO machines
  - map AO core mechanisms to OO mechanisms
  - AO works on target machine instead of being supported in it

- OO languages have dedicated target machines
  - Java, Smalltalk, C++ (run-time implementation), ...
  - OO machines support OO core mechanisms
Weaver Targets

• commonalities of weaving approaches
  – all target non-AO machines
  – map AO core mechanisms to OO mechanisms
  – AO works on target machine instead of being supported in it

• OO languages have dedicated target machines
  – Java, Smalltalk, C++ (run-time implementation), ... 
  – OO machines support OO core mechanisms

Where is the aspect machine?
Language mechanisms deserve language implementation effort.
Weaver Targets

- commonalities of weaving approaches
  - all target non-AO machines
  - map AO core mechanisms to OO mechanisms
  - AO works on target machine instead of being supported in it

- OO languages have dedicated target machines
  - Java, Smalltalk, C++ (run-time implementation), ...
  - OO machines support OO core mechanisms

Where is the aspect machine?
Language mechanisms deserve language implementation effort.
Core AO Mechanisms

- a different look at join points
  - at each join point, decisions are drawn
  - which functionality to execute?
  - depends on context (current object, message sent/received, methods on stack, ...)

- very much like polymorphic dispatch
Core AO Mechanisms

- a different look at join points
  - at each join point, decisions are drawn
  - which functionality to execute?
  - depends on context (current object, message sent/received, methods on stack, ...)
- very much like polymorphic dispatch
- running application ≈ sequence of late binding events
  - a join point is a locus of late binding
  - like virtual methods: virtual join points
Core AO Mechanisms

- a different look at join points
  - at each join point, decisions are drawn
  - which functionality to execute?
  - depends on context (current object, message sent/received, methods on stack, ...)
- very much like polymorphic dispatch
- running application ≈ sequence of late binding events
  - a join point is a locus of late binding
  - like virtual methods: virtual join points
A Machine Model for AOP

- core concepts: objects and delegation
  - each application object, internally, consists of two
  - a proxy (a mere placeholder) and the actual object
  - other application objects only directly see the proxy
  - access to properties only via messages

```
object delegate actual-object

foo = (...)
bar = 23
```
A Machine Model for AOP

- core concepts: objects and delegation
  - each application object, internally, consists of two
  - a proxy (a mere placeholder) and the actual object
  - other application objects only directly see the proxy
  - access to properties only via messages

- classes and instances
  - classes are also represented by objects, also have a proxy
  - instances reference the proxy only
  - delegation ensures correct binding of `self`

Michael Haupt, Hans Schippers (michael.haupt@hpi.uni-potsdam.de, hans.schippers@ua.ac.be)
A Machine Model for AOP

- core concepts: objects and delegation
  - each application object, internally, consists of two
  - a proxy (a mere placeholder) and the actual object
  - other application objects only directly see the proxy
  - access to properties only via messages

- classes and instances
  - classes are also represented by objects, also have a proxy
  - instances reference the proxy only
  - delegation ensures correct binding of self
A Machine Model for AOP

- core concepts: objects and delegation
  - each application object, internally, consists of two
  - a proxy (a mere placeholder) and the actual object
  - other application objects only directly see the proxy
  - access to properties only via messages

- classes and instances
  - classes are also represented by objects, also have a proxy
  - instances reference the proxy only
  - delegation ensures correct binding of self
A Machine Model for AOP

- core concepts: objects and delegation
  - each application object, internally, consists of two
  - a proxy (a mere placeholder) and the actual object
  - other application objects only directly see the proxy
  - access to properties only via messages

- classes and instances
  - classes are also represented by objects, also have a proxy
  - instances reference the proxy only
  - delegation ensures correct binding of self
// CaesarJ-like pseudo code

class C {
    void f() { ... }
    void g() { ... }
    void h() { ... }
}

class A {
    before():
        execution(void C.f())
        { ... }
    after():
        execution(void C.g())
        { ... }
    void around():
        execution(void C.h())
        { ... proceed(); ... }
}
Basic Weaving Operations

// CaesarJ-like pseudo code

class C {
    void f() { ... }
    void g() { ... }
    void h() { ... }
}

class A {
    before():
        execution(void C.f())
        { ... }
    after():
        execution(void C.g())
        { ... }
    void around():
        execution(void C.h())
        { ... proceed(); ... }
}

... deploy(new A()); ...
Basic Weaving Operations

// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... }
    void h() { ... }
}

class A {
    before():
        execution(void C.f())
        { ... }
    after():
        execution(void C.g())
        { ... }
    void around():
        execution(void C.h())
        { ... proceed(); ... }
}

... deploy(new A()); ...

instance-of-C

actual-instance-of-C

A-proxy
f = (...; resend )
g = ( resend; ... )
h = (...; resend; ... )
Instance-Local Deployment

// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... }
    void h() { ... }
}

class A {
    before():
        execution(void C.f())
        { ... }
    after():
        execution(void C.g())
        { ... }
    void around():
        execution(void C.h())
        { ... proceed(); ... }
}

... deploy(new A(),
    instance-of-C);

actual-instance-of-C

C

instance-of-C

actual-C

f = (...)
g = (...)
h = (...)

---

Michael Haupt, Hans Schippers (michael.haupt@hpi.uni-potsdam.de, hans.schippers@ua.ac.be)
Instance-Local Deployment

// CaesarJ-like pseudo code

class C {
    void f() { ... }
    void g() { ... }
    void h() { ... }
}

class A {
    before():
        execution(void C.f())
        { ... }
    after():
        execution(void C.g())
        { ... }
    void around():
        execution(void C.h())
        { ... proceed(); ... }
}

... deploy(new A(),
    instance-of-C); ...
Thread-Local Deployment

// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... }
    void h() { ... }
}

class A {
    before():
        execution(void C.f())
        { ... }
    after():
        execution(void C.g())
        { ... }
    void around():
        execution(void C.h())
        { ... proceed(); ... }
}

// in thread T1
... deploy(new A()) { ... } ...
Thread-Local Deployment

// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... }
    void h() { ... }
}

class A {
    before():
        execution(void C.f())
        { ... }
    after():
        execution(void C.g())
        { ... }
    void around():
        execution(void C.h())
        { ... proceed(); ... }
}

// in thread T1
... deploy(new A()) { ... } ...

actual-instance-of-C

actual-C
f = (...)
g = ( resend; ... )
h = ( ...; resend; ... )

instance-of-C
Thread-Local Deployment

// CaesarJ-like pseudo code
class C {
  void f() { ... }
  void g() { ... }
  void h() { ... }
}

class A {
  before():
    execution(void C.f())
    { ... }
  after():
    execution(void C.g())
    { ... }
  void around():
    execution(void C.h())
    { ... proceed(); ... }
}

// in thread T1
... deploy(new A()) { ... } ...

instance-of-C

actual-instance-of-C

the delegate of C is a function of the thread

actual-C
f = (...) 
g = ( resend; ...) 
h = ( ...; resend; ... )
cflow: Continuous Weaving

// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... f(); ... }
}

class A {
    before():
        execution(void C.f()) &&
        cflow(execution(void C.g()))
        { ... }
}
cflow: Continuous Weaving

\[ \text{deploy(new } A()); \ldots \]

// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... f(); ... }
}

class A {
    before():
        execution(void C.f()) &&
        cflow(execution(void C.g()))
        { ... }
}
cflow: Continuous Weaving

```java
// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... f(); ... }
}

class A {
    before():
        execution(void C.f()) &&
        cflow(execution(void C.g()))
        { ... }
}
```

```
instance-of-C

actual-instance-of-C

C

A-cw-proxy

g = (act-cflow; resend; deact-cflow)

... deploy(new A()); ...
```
cflow: Continuous Weaving

// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... f(); ... }
}

class A {
    before():
        execution(void C.f()) &&
        cflow(execution(void C.g()))
        { ... }
}

... deploy(new A()); ...
// in thread T1
... instance-of-C.g(); ...

instance-of-C

actual-instance-of-C

C

A-cw-proxy

g = ( act-cflow; resend; deact-cflow )

actual-C
f = (... )
g = (... )
cflow: Continuous Weaving

// CaesarJ-like pseudo code
class C {
  void f() { ... }
  void g() { ... f(); ... }
}

class A {
  before():
    execution(void C.f()) &&
    cflow(execution(void C.g()))
    { ... }
}

... deploy(new A()); ...
// in thread T1
... instance-of-C.g(); ...

A-proxy
f = (...; resend)

A-cw-proxy
g = (act-cflow; resend; deact-cflow)

actual-C
f = (...)
g = (...)

instance-of-C

actual-instance-of-C

C
T1

others
cflow: Continuous Weaving

// CaesarJ-like pseudo code
class C {
    void f() { ... }
    void g() { ... f(); ... }
}

class A {
    before():
        execution(void C.f()) &&
        cflow(execution(void C.g()))
        { ... }
}

... deploy(new A()); ... // in thread T1
... instance-of-C.g(); ...
// CaesarJ-like pseudo code

class C {
    int x;
    void f() { ... }
}

class A {
    int C.y;
    void C.g() { ... }
    before():
        execution(void C.f())
        { ... }
}

actual-instance1-of-C
  x = 23
  instance1-of-C

actual-instance2-of-C
  x = 42
  instance2-of-C

C

actual-C
  f = (...)

Michael Haupt, Hans Schippers (michael.haupt@hpi.uni-potsdam.de, hans.schippers@ua.ac.be)
Introduction

// CaesarJ-like pseudo code

class C {
    int x;
    void f() { ... }
}

class A {
    int C.y;
    void C.g() { ... }
    before():
        execution(void C.f())
        { ... }
}

... deploy(new A()); ...
// CaesarJ-like pseudo code

class C {
    int x;
    void f() { ... }
}

class A {
    int C.y;
    void C.g() { ... }
    before():
        execution(void C.f())
        { ... }
}

... deploy(new A()); ...

A-int-proxy
y = (/* add y field to instance */; self y )
g = (/* imp. of introduced method */)

actual-C
f = (...)

C

actual-instance1-of-C
x = 23

instance1-of-C

actual-instance2-of-C
x = 42

instance2-of-C
Introduction

// CaesarJ-like pseudo code
class C {
    int x;
    void f() { ... }
}

class A {
    int C.y;
    void C.g() { ... }
    before():
        execution(void C.f())
        { ... }
}

... deploy(new A()); ...
... instance1-of-C.y ...

instance1-of-C
actual-instance1-of-C
x = 23

instance2-of-C
actual-instance2-of-C
x = 42

C
A-int-proxy
y = ( /* add y field to instance */; self y )
g = ( /* imp. of introduced method */ )

actual-C
f = (...)

Michael Haupt, Hans Schippers (michael.haupt@hpi.uni-potsdam.de, hans.schippers@ua.ac.be)
Introduction

// CaesarJ-like pseudo code
class C {
    int x;
    void f() { ... }
}

class A {
    int C.y;
    void C.g() { ... }
    before() {
        execution(void C.f())
        { ... }
    }
}

... deploy(new A()); ...
... instance1-of-C.y ...

\[
\begin{align*}
\text{actual-instance1-of-C} & : x = 23 \\
\text{instance1-of-C} & : x = 23 \\
\text{instance2-of-C} & : x = 42 \\
\text{actual-instance2-of-C} & : x = 42 \\
\end{align*}
\]

\[
\begin{align*}
C & \quad \text{A-int-proxy} \\
A-int-proxy & : y = (/* add y field to instance */; self y) \\
g & = (/* imp. of introduced method */)
\end{align*}
\]
// CaesarJ-like pseudo code
class C {
    int x;
    void f() { ... }
}

class A {
    int C.y;
    void C.g() { ... }
    before():
        execution(void C.f())
            { ... }
}

... deploy(new A()); ...
... instance1-of-C.y ...

actual-instance1-of-C
x = 23

actual-instance2-of-C
x = 42

actual-C
f = (...)

A-int-proxy
y = ( /* add y field to instance */; self y )
g = ( /* imp. of introduced method */ )
A Machine Model for Aspect-Oriented Programming

Introduction

// CaesarJ-like pseudo code
class C {
    int x;
    void f() { ... }
}

class A {
    int C.y;
    void C.g() { ... }
    before():
        execution(void C.f())
        { ... }
}

... deploy(new A()); ...
... instance1-of-C.y ...

A-int1-proxy
y = 0

A-int-proxy
y = (/* add y field to instance */; self y )
g = (/* imp. of introduced method */)

actual-instance1-of-C
x = 23

actual-instance2-of-C
x = 42

instance1-of-C

instance2-of-C

C

Michael Haupt, Hans Schippers (michael.haupt@hpi.uni-potsdam.de, hans.schippers@ua.ac.be)
// CaesarJ-like pseudo code

class C {
    int x;
    void f() { ... }
}

class A {
    int C.y;
    void C.g() { ... }
    before():
        execution(void C.f())
            { ... }
}

... deploy(new A()); ...
... instance1-of-C.y ...

A-int1-proxy
y = 0

instance1-of-C

Y

actual-instance1-of-C
x = 23

actual-instance2-of-C
x = 42

instance2-of-C

C

A-int-proxy

y = ( /* add y field to instance */; self y )
g = ( /* imp. of introduced method */ )

actual-C
f = ( ... )

Michael Haupt, Hans Schippers (michael.haupt@hpi.uni-potsdam.de, hans.schippers@ua.ac.be)
Formalisation

- \( \delta \) calculus
  - imperative, object-based with delegation
  - operational semantics:

  \[
  \text{(Select)} \quad a, \xrightarrow{} \boxed{\text{x}} \xrightarrow{} \boxed{\text{x}}
  \]
  \[
  \text{Look}(\boxed{\text{x}}, \boxed{\text{m}}) = \{b\}
  \]
  \[
  \boxed{\text{x}} = \boxed{\text{x}} [\text{self } \rightarrow \boxed{\text{x}}]
  \]
  \[
  b, \xrightarrow{} \boxed{\text{x}} \xrightarrow{} \boxed{\text{x}}
  \]
  \[
  \boxed{\text{x}}' = \boxed{\text{x}} [\text{self } \rightarrow \boxed{\text{self}}]
  \]

  \[
  a.m, \xrightarrow{} \boxed{\text{x}} \xrightarrow{} \boxed{\text{x}}'
  \]

- extended with aspect-oriented support:

  \[
  \text{(Deploy Aspect)} \quad a, \xrightarrow{} \boxed{\text{x}} \xrightarrow{} \boxed{\text{x}} \xrightarrow{} \boxed{\text{x}}
  \]
  \[
  \text{asp}, \xrightarrow{} \boxed{\text{y}} \xrightarrow{} \boxed{\text{y}}
  \]
  \[
  \boxed{\text{y}} = \boxed{\text{y}} \text{Del}_\text{asp}(\text{context}) = \text{Del}_\text{asp}
  \]
  \[
  \boxed{\text{y}}' = \boxed{\text{y}} [\text{Del}_\text{asp}(\text{context}) = \boxed{\text{y}}]
  \]

  \[
  \text{deploy(asp,a), } \boxed{\text{x}} \xrightarrow{} \boxed{\text{x}} \xrightarrow{} \boxed{\text{x}}'
  \]
Formalisation

- $\delta$ calculus
  - imperative, object-based with delegation
  - operational semantics:

  \[
  \begin{align*}
  & (Select) \\
  & a, \xrightarrow{x} x \times x \times \\
  & \text{Look}(x \times x \times m) = \{b\} \\
  & x = x \times \text{self} \rightarrow x \times y \\
  & b, \xrightarrow{y} y \times y \times y' = y \times [\text{self} \rightarrow x \times \text{self}] \\
  & a.m, \xrightarrow{m} x \times y' \\
  & \text{extended with aspect-oriented support:}
  \end{align*}
  \]

  \[
  \begin{align*}
  & (Deploy \text{ Aspect}) \\
  & a, \xrightarrow{x} x \times x \times x' \\
  & \text{asp}, \xrightarrow{a} x \times x \times x \\
  & x = x \times \text{Del}_x (\text{context}) = \text{Del}_x \\
  & x' = x \times [\text{Del}_x (\text{context}) = x] \\
  & \text{deploy}(\text{asp}, a), \xrightarrow{a} x \times x \times x' 
  \end{align*}
  \]
Future Work

- shape of an aspect-oriented machine
  - for example, for Java: looks like an "ordinary" JVM
  - apart from one extra bytecode or two (for aspect weaving)
Future Work

- shape of an aspect-oriented machine
  - for example, for Java: looks like an "ordinary" JVM
  - apart from one extra bytecode or two (for aspect weaving)

- performance
  - potential problem: high amount of message sends
  - optimisation of polymorphic message sends is very well understood (polymorphic inline caches etc.)
Future Work

- shape of an aspect-oriented machine
  - for example, for Java: looks like an "ordinary" JVM
  - apart from one extra bytecode or two (for aspect weaving)

- performance
  - potential problem: high amount of message sends
  - optimisation of polymorphic message sends is very well understood (polymorphic inline caches etc.)
Future Work

• aspect machine implementations
  – prototypical implementation in Io exists
  – currently being addressed
  – experimental JVM with core AOP support
  – based on Id/Pepsi/Jolt
Future Work

- aspect machine implementations
  - prototypical implementation in lo exists
  - currently being addressed
  - experimental JVM with core AOP support
  - based on Id/Pepsi/Jolt

- AOP language mappings
  - formalisation (extension to $\delta$ calculus) exists
  - take core language semantics, map to model semantics
  - run on semantics implementation (Jolt)
  - compare to weaving into Java-like AST
Future Work

- aspect machine implementations
  - prototypical implementation in \texttt{Io} exists
  - currently being addressed
  - experimental JVM with core AOP support
  - based on \texttt{Id/Pepsi/Jolt}

- AOP language mappings
  - formalisation (extension to $\delta$ calculus) exists
  - take core language semantics, map to model semantics
  - run on semantics implementation (\texttt{Jolt})
  - compare to weaving into Java-like AST
Summary

- aspect-oriented language implementations
  - still mostly mappings to object-oriented mechanisms
  - AOP is a paradigm in its own right

- model of an "aspect machine"
  - core mechanism: late binding at join points
  - achieved using proxies and delegation
  - expressive enough for current AOP languages

- aspect machine implementations
  - not necessarily fully new machines
  - superset of existing implementations
  - AOP support at their core