Language support for AOP

*AspectJ and beyond*

Mario Südholt

www.emn.fr/sudholt

INRIA and École des Mines de Nantes

OBASCO project, Nantes, France
Plan

• AOP and AspectJ
• AspectJ: end of story?
• Beyond AspectJ
I. AOP and AspectJ
Defining characteristics of AOP?

- **Quantification**: modularization of crosscutting concerns
- **Obliviousness**: non-anticipation; incremental development

⇒ Tackle crosscutting in large-scale applications throughout the software life cycle

More probably later from Bob . . .
What’s new?  (1)

• What about **computational reflection**?
  • 3-Lisp, CLOS, Reflex [Tanter et al., OOPSLA’03], . . .
  • General enough reflective system can “emulate” AOP systems
• Difficult to understand
• Performance issues
• Semantics issues, lack of correctness guarantees
What’s new? (2)

• What about transformation systems?
  • General enough transformation system can “emulate” AOP systems
  • SOOT, Recoder, CIL, . . .
  • Difficult to understand
  • Correctness properties difficult to handle
Yes, it is! (in a sense)

Goals for AOP

- Provide abstractions general enough to modularize (some or all) concerns.
- Be specific enough to make such modularization understandable, tractable and amenable to testing, analysis, verification of properties.
AspectJ in one slide

Base program: critical, access
**AspectJ in one slide**

```
pointcut critAcc(Base r): call(void Base.acc(int)
    && target(r)
    && cflow(call(void Base.crit(int))));
```
**AspectJ in one slide**

```java
pointcut critAcc(Base r): call(void Base.acc(int)
    && target(r)
    && cflow(call(void Base.crit(int))));

void around(Base r):   _critAcc(r) {
    calls++;
    if (ok()) proceed(r);
}
```
AspectJ in one slide

```java
aspect ProfBar pertarget call(void Base.acc(int)) {
    int calls = 0;

    pointcut critAcc(Base r): call(void Base.acc(int)
        && target(r)
        && cflow(call(void Base.crit(int))));

    void around(Base r): critAcc(r) {
        calls++;
        if (ok()) proceed(r);
    }
}
```

“Language support for AOP”; Mario Südholt; INRIA/EMN; March 8, 2005 – p.8/28
aspect ProfBar pertarget call(void Base.acc(int)) {
    int calls = 0;
    static int Base.calls = 0;

    pointcut critAcc(Base r): call(void Base.acc(int)
        && target(r)
        && cflow(call(void Base.crit(int))));

    void around(Base r): critAcc(r) {
        calls++;
        if (ok()) proceed(r);
    }
}
Characteristics of AspectJ

+ Join points
+ Pointcuts
+ Advice
+ Aspects
+ Inter-type declarations

← Aspect instantiation (coarse-grained)
← Aspect activation (on/off)
← Aspect composition (dominate)
II. AspectJ: end of story?

- Other characteristics of aspect languages
- Other base languages, execution environments
- More expressive pointcut languages
Other characteristics of aspect languages

• Aspect instantiation
  E.g., runtime instances, Kevin’s talk

• Aspect activation
  E.g., enable/disable aspects at runtime

• Aspects of aspects
  E.g., layered aspects, Kevin’s talk

• Aspect composition
  E.g. for conflict resolution

• Weaver semantics
  E.g., no aspects of aspects
A world outside Java?

- Crosscutting concerns in large (legacy) C applications
- Ex.: optimization of web caches without cache flushes
- New aspect languages for expression of complex context conditions
Other pointcut languages  (1)

- **Stateful pointcuts** (explicit state in pointcuts)
  - Sequence pointcuts:
    Ex.: protocol translation and bug correction
  - Temporal logic pointcuts:
    Ex.: manipulation of Linux kernel code
  - Regular expression pointcuts:
    Enable interference analysis among aspects
Other pointcut languages (2)

- AOP and distributed applications
  - Often integration/configuration of existing distribution platforms (see Kevin’s talk)
    ⇒ distribution implicit to aspects
- **Remote pointcuts** [Nishizawa et al., AOSD’04]: explicit hosts, advice server
  - Trade-off: hide complexity vs. flexibility
- **Data-flow pointcuts** [Masuhara, Kawauchi; APLAS’03], e.g., for security enforcement.
  Efficiency realization
III. Beyond AspectJ

1. Dynamic aspects for C system-level applications
2. Temporal logic pointcuts for Linux kernel evolution
1. Dynamic aspects for C system-level applications

- Software evolution frequently to be performed on running systems (e.g., high-availability servers)
- Ex. concerns in a web cache
  - Modification of caching policies
  - Optimizations (e.g., protocol transformations TCP→UDP)
  - Bug corrections
- Some large applications:
  Open-source web-cache “squid”: 9 MB of source
Ex.: explicit sequences for buffer overflows

• Aspect language with explicit sequences

\[
\text{seq}( \text{call}(\text{void } \ast \text{malloc(size\_t)})) \\
\quad \text{&& args(allocatedSize) && return(buffer);} \\
\quad \text{write(buffer) && size(writtenSize)} \\
\quad \text{&& if(writtenSize > allocatedSize)} \\
\quad \text{then reportOverflow(); } \ast \\
\quad \text{call(\text{void free(\text{void}*)})}
\]
Aspect language

• Primitive pointcuts: calls and variables accesses (to global and local variables)
• cflow for nested calls (like AspectJ)
• Sequences with
  • Conditionals over data
    Principally equalities (e.g. over file handles)
  • Means for ressource handling
    Optimize ressource usage (e.g., reuse of file handles)
Realization: the Arachne system

- Dynamic aspect application for C without program interruption
  www.emn.fr/x-info/arachne

- Rewrite binary code on the fly to weave (and deweave) aspects

- Current weaving semantics excludes nested aspects
  Simplified implementation, somewhat more efficient

- [Ségura et al, AOSD’03] [Fritz et al, AOSD’05]
2. Temporal logic pointcuts for Linux kernel evolution

• Problem: support extensions of the Linux kernel over a range of kernel versions
  E.g., over one major version number

• Ex.: support application-specific schedulers
  E.g., for multi-media streaming

• Context: integrate an existing system for scheduler development with the kernel
Bossa: new schedulers for plain old Linux

- **Bossa**: system for scheduler development
  www.emn.fr/x-info/bossa
- **DSL**: definition of scheduling policies
- Support runtime for hierarchical, prioritized, etc., schedulers
- Runtime overhead < 5%
Bossa architecture

- Events mediate between (instrumented) kernel and Bossa runtime, which supports policies.
Mediation through events crosscuts the kernel

- Instrument kernel code + drivers (~ 100 MB source code)
- Instrumentation for Bossa:
  - ~ 400 instructions changed in about 150 files
- Previously manually done for Linux kernel 2.4
- Can we do better with aspects?
Problem: context dependencies

- Generate events for \texttt{schedule} instructions
- Other instructions relevant (e.g., thread state, \texttt{yield})
- Problem: thread context implicit
- Explicit context dependencies vs. efficiency
Solution: temporal logic pointcuts

- Use temporal predicates to express control-flow relationships

\[ n : \text{Rewrite}(n, \text{schedule}\_\text{running}) \]

If \( n \vdash AX \triangle (A \triangle (\neg \text{changeOfState}() \cup \text{changeToRunning}())) \)

“Change current instruction to \text{schedule}\_\text{running} if for all backward pathes starting from the predecessor node, all backward pathes change to running without previous changes to the state.”
Results

• Transformational system for Bossa integration: 25 rules
• Implementation based on CIL yields exact instrumentation
  ⇒ no overhead to manual instrumentation
• 6 bugs of manual instrumentation detected
• [Åberg et al., ASE’03]
Conclusion

• AOP is relevant to software development
• AOP interesting from theoretical and practical viewpoint
• AspectJ is an interesting language and tool but not the end of the story
Future work

• (Almost) everything still to be done
• AOP for distributed programming
  • Remote pointcut: extend language, implementation, remote aspect calculus
• Aspect interactions
  • Generalize first results over regular expressions, use of model checking
• Aspects and components
  • Aspects over components with explicit protocols