Konfigurierbare Systemsoftware (KSS)

VL 4 – Aspect-Aware Development: The CiAO Approach

Daniel Lohmann
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg
SS 14 – 2014-05-08

http://www4.informatik.uni-erlangen.de/Lehre/SS14/V_KSS

About this Lecture

Implementation Techniques: Classification

- Decompositional Approaches
  - Text-based filtering (untyped)
  - Preprocessors
- Compositional Approaches
  - Language-based composition mechanisms (typed)
  - OOP, AOP, Templates
- Generative Approaches
  - Metamodel-based generation of components (typed)
  - MDD, C++ TMP, generators

Implementation Techniques: Goals

General
- Separation of concerns (SoC)
- Resource thriftiness

Operational
- Granularity
  Components should be fine-grained. Each artifact should either be mandatory or dedicated to a single feature only.
- Economy
  The use of memory/run-time expensive language features should be avoided as far as possible. Decide and bind as much as possible at generation time.
- Pluggability
  Changing the set of optional features should not require modifications in any other part of the implementation. Feature implements should be able to “integrate themselves”.
- Extensibility
  The same should hold for new optional features, which may be available in a future version of the product line.
**Agenda**

4.1 AOP Mechanisms Under the Hood
4.2 Study: i4Weathermon AOP
4.3 CiAO
4.4 CiAO Results
4.5 Summary
4.6 References

---

**Agenda**

4.1 AOP Mechanisms Under the Hood

**Diagram Notation**

- Obliviousness & Quantification
- AOP Mechanisms: Summary

4.2 Study: i4Weathermon AOP
4.3 CiAO
4.4 CiAO Results
4.5 Summary
4.6 References

---

**Notation**

```
BaseClass
method()

explicitJP()•
method is explicit join point
```

```
SomeClass
protected()
static()
unadvisable()
```

```
AnAspect
exec("explicitJP")
intro("SomeClass")
```

```
<slice> ASlice
state_variable
anotherMethod()
```

---

**AOP Mechanisms Demystified: "Obliviousness"**

**Scenario:**
Optional feature component $F_1$ shall be integrated into SPL component $PL$

- With OOP:
  - $PL$ has to call $F_1 \rightarrow PL$ has to know $F_1$
  - control flows can only be established in the direction of knowledge

- With AOP:
  - $F_1$ can give advice to $PL \sim F_1$ has to may know $PL$
  - control flow is established opposite to the direction of knowledge
  - binding is inherently loose $\sim$ silently missed, if $PL$ does not exist

---
AOP Mechanisms Demystified: “Quantification”

**Scenario:**
(Nonfunctional)
feature component $F_1$ shall be integrated into (optional) SPL components $PLC_{1...n}$

- With AOP:
  - binding is inherently loose $\leadsto$ may quantify over $n$ join points
  - possible by delarative pointcut concept
    (here: wildcard in match expression)

- Advice inverses the direction in which control-flow relationships are established:
  $C$ calls $A \implies A$ advises $C$

- Aspects integrate themselves into the surrounding program $\leadsto$ “I make you call me”
- Surrounding program can be kept oblivious of the aspects $\leadsto$ advice-based binding as a means to integrate (optional) features

- Pointcuts provide for an implicit quantification of this integration
  - Applies to $0 \ldots M \ldots n$ join points, depending on the pointcut expression
  - Thereby, advice-based binding is inherently loose $\leadsto$ advice-based binding as a means to integrate interacting features

---

Agenda

4.1 AOP Mechanisms Under the Hood
4.2 Study: i4Weathermon AOP
  - Flashback: i4Weathermon
  - i4WeatherMon with AOP
  - i4WeatherMon with AOP: Results
4.3 CiAO
4.4 CiAO Results
4.5 Summary
4.6 References

---

I4WeatherMon: Feature Model

- How to achieve **Granularity, Economy, Pluggability, Extensibility?**
  - Configuration-dependent sensor and actuator sets
    - initialization, integration, interaction of optional feature code
  - Generic and nongeneric actuators
    - interacting optional feature code
### I4WeatherMon: OOP Solution Space

#### Basic structure
- **Weather** and **Sink** are (almost) empty classes.
  - Provide a **lexical scope** for sensor/actuator introductions
  - Provide **explicit join points** (empty methods `measure()` / `process()`) that are invoked by the main loop, when measuring/processing should take place
  - All further functionality is provided by the aspects!

### I4WeatherMon: AOP Solution Space

#### Pressure Handling
- **Pressure**
  - `measure()`
  - `exec("Weather::measure()")` into `Weather`

#### Wind Handling
- **Wind**
  - `measure()`
  - `exec("Weather::measure()")` into `Weather`

#### Sink
- `exec("Sink::process_data(Temperature)")`
- `exec("Sink::process()")`
- `exec("Sink::after_process()")`
- `exec("Sink::before_process()")`

#### Display
- `#print()`
- `#send()`

#### Sensor integration
- A Sensor is implemented as a class with an accompanying Handling aspect
  - Slices the sensor singleton instance into **Weather**
  - Gives advice to **Weather::measure()** to invoke **Sensor::measure()**
  - Slices an **explicit join point** `process_data(Sensor)` into Sink
  - Gives advice to **Sink::process()** to invoke `process_data(Sensor)`
I4WeatherMon: AOP Sensor Integration

class Weather {
public:
    void measure() {} // empty implementation
};

class Sink {
public:
    void process() {} // empty implementation
};

aspect PressureHandling {
    // Weather integration
    advice "Weather" : slice struct {
        Pressure _pressure; // introduce sensor instance (singleton)
    };
    advice execution("void Weather::measure()") : before() {
        theWeather.pressure.measure(); // invoke sensor’s measure()
    } // Sink integration
    advice "Sink" : slice struct {
        // introduce sensor-specific explicit join point for actuator aspects
        void process_data( const Pressure & ) {};
    };
    advice execution("void Sink::process()") : after() {
        theSink.process_data( theWeather.pressure ); // trigger it
    }
};

I4WeatherMon: AOP Generic Actuator Integration

aspect Display {
    // display each element of the weather data
    advice execution("void Sink::process_data(%)") : before() {
        typedef JoinPoint<template Arg<0>::ReferredType Data; char val[];
        tjp->arg<0>()->str_val( val );
        print( Data::name(), val, Data::unit() );
    }
};

I4WeatherMon: AOP Solution Space

Generic actuator integration

A generic actuator (processes all sensors) is implemented by an aspect
- Gives advice to Sink::process() to execute processing pre-/post actions
- Gives generic advice to all overloads of Sink::process_data() to invoke each sensor (typed) in order to process its data via the generic str_val()

~ Generic actuator does not know the available / possible sensor types

Nongeneric actuator integration

A nongeneric actuator (processes some sensors) is implemented by an aspect
- Gives advice to Sink::process() to execute processing pre-/post actions
- Gives advice to selected overloads of Sink::process_data() to invoke them in order to process each sensors data via a sensor-specific interface

~ Nongeneric actuator has to know specific sensor types
I4WeatherMon: AOP Generic Actuator Integration

```java
aspect SNGConnection : protected PCConnection {
    UInt8 _p, _w, _t1, _t2; // weather record

    // let this aspect take a higher precedence than <Sensor>Handling
    advice process () : order("SNGConnection", "<Handling>")
        : before () { ... /* init record */ } 
        : after () { ... /* transmit record */ } 

    // collect wind, pressure, temperature data by giving specific advice
    advice execution("void Sink::process(const Weather&)") && args(wind)
        : before (const Wind &wind) {
            _w = wind._w;
        }
    advice execution("void Sink::process(const Weather&)") && args(pressure)
        : before (const Pressure &pressure) {
            _p = pressure._p - 850;
        }
    advice execution("void Sink::process(const Weather&)") && args(temp)
        : before (const Temperature &temp) {
            _t1 = (UInt8)temp._t1;
            _t2 = temp._t2;
        }
};
```

I4WeatherMon (AOP): Evaluation

General
- Separation of concerns (SoC)
- Resource thriftiness

Operational
- Granularity
  - Every component implements functionality of a single feature only.
- Economy
  - All control-flow bindings are established at compile time.
- Pluggability
  - Sensors and actuators integrate themselves by aspects.
- Extensibility
  - “Plug & Play” of sensor and actuator implementations.

Agenda

4.1 AOP Mechanisms Under the Hood
4.2 Study: i4Weathermon AOP
4.3 CiAO
  - Motivation and Goals
  - Design Approach
  - Examples: Aspects in Action
  - Further Examples
4.4 CiAO Results
4.5 Summary
4.6 References
CiAO: Motivation and Goals

Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details (mechanisms). This separation allows maximum flexibility if policy decisions are to be changed later. 

Silberschatz, Gagne, and Galvin 2005: Operating System Concepts [8, p. 72]

- Primary goal: architectural configurability
  - configurability of even fundamental policies
    - synchronization, protection, interaction
- Secondary goal: <the standards>
  - efficiency, configurability in general, portability
- Approach: aspect-aware operating system design
  - strict decoupling of policies and mechanisms in the implementation

CiAO: General Structure

Layered Architecture
- Interface layer (as/ciao)
- System layer (os)
- Hardware layer (hw)

Layers \(\rightarrow\) C++ namespaces
- Potential join points for cross-layer transitions
- Further refined by sublayers (os::krn, hw::irq)
- Layers as a means of aspect-aware development

Methodology: Principles of Aspect-Aware Development

Design Principles \(\rightarrow\) Development Idioms
1. loose coupling \(\rightarrow\) advice-based binding
2. visible transitions \(\rightarrow\) explicit join points
3. minimal extensions \(\rightarrow\) extension slices
Methodology: Principles of Aspect-Aware Development

The principle of loose coupling. Make sure that aspects can hook into all facets of the static and dynamic integration of system components. The binding of components, but also their instantiation (e.g., placement in a certain memory region) and the time and order of their initialization should all be established (or at least be influenceable) by aspects.

The principle of visible transitions. Make sure that aspects can hook into all control flows that run through the system. All control-flow transitions into, out of, and within the system should be influenceable by aspects. For this they have to be represented on the join-point level as statically evaluable, unambiguous join points.

The principle of minimal extensions. Make sure that aspects can extend all features provided by the system on a fine granularity. System components and system abstractions should be fine-grained, sparse, and extensible by aspects.

Methodology: Roles of Aspects and Classes

What to model as a class and what as an aspect?

- `<Thing>` is a class if – and only if – it is a distinguishable, instantiable concept of CiAO:
  - A system component, instantiated internally on behalf of CiAO
    - The Scheduler, the Alarm Manager, the OS control facility, ...
    - Hold and manage kernel state, singletons by definition
  - A system abstraction, instantiated as objects on behalf of the user
    - Task, Event, Resource, ...
    - In AUTOSAR OS: instantiated at compile time
  - Both are sparse — provide a minimal implementation only
- Otherwise `<thing>` is an aspect!

Example: Mechanism Integration

---

Three idiomatic aspect roles

- Extension aspects: extend some system component or system abstraction by additional functionality.
- Policy aspects: “glue” otherwise unrelated system abstractions or components together to implement some CiAO kernel policy.
- Upcall aspects: bind behavior defined by higher layers to events produced in lower layers of the system.
Example: Mechanism Integration

**Binding Aspects**
- Implement upcalls by advice-based bindings

```
aspect Serial0Ext {
    advice execution( void hw::hal::init() ) :
        order( "Sched_Init", "Serial0_Init" );
};
```

Example: Mechanism Integration

**Policy Aspects**
- Add to mechanisms by extension slices

```
{\textit{policy aspect}}
    Kernel_ASTSync
    exec("enterKernel()")
    exec("leaveKernel()")

{\textit{policy aspect}}
    Sched_ASTBinding
    exec("ast")
    trigger()
    disable()
    enable()
```

Example: Mechanism Integration

```
aspect Serial0Ext {
    advice execution( void hw::hal::init() ) :
        order( "Sched_Init", "Serial0_Init" );
};
```
Example: Policy Integration

Cooperative Scheduling:
- `enterKernel()`
- `leaveKernel()`

Preemptive Scheduling:
- `enterKernel()`
- `leaveKernel()`

- `policy aspect: Sched_LeaveBinding`
- `exec("leaveKernel")`

- `policy aspect: Kernel_ASTSync`
- `exec("setNeedReschedule")`
- `exec("ast")`

Methodology: Explicit Join Points

- Advice-based binding → availability of the "right" join points
  - for all semantically important transitions in the system
  - statically evaluable

- Fine-grained component structure ~ many implicit join points, but
  - amount and precise semantics often implementation dependent
  - aspects have to "know" ~ no obliviousness

- Important transitions not available for technical reasons as JPs
  - target code may be fragile (e.g., context switch) ~ must not be advised
  - target code may be written in assembly ~ transitions not visible as JPs

Solution: explicit join points
- empty inline methods for the sole purpose that aspects can bind to them
  - explicitly triggered by components or other aspects
  - well defined semantics

Upcall join points (U) represent system-internal events that are to be processed on a higher layer
- exceptions, such as signals or interrupts
- internal events, such as system initialization or entering of the idle state

Transition join points (T) represent semantically important control-flow transitions inside the kernel
- level transitions: `user → kernel`, `user → interrupt`
- context transitions: `threadA → threadB`
Example: Optional Feature Interaction

Explicit join points for the support and binding of OSEK OS and AUTOSAR OS user-level hook functions, as specified in [36, p. 39] and [4, p. 46]. Triggered in case of an error, a protection violation, before (pre) and after (post) at high-level task switch, and at operating system startup and shutdown time.

Typically triggered when a control flow enters (respectively leaves) the kernel domain.

Entry point of a new thread (continuation).

Entry point of the respective AST.

Entry point of the respective interrupt handler. (Interrupts are still disabled.)

Triggered immediately before the running continuation is deactivated or terminated; to is going to become the next running continuation.

Triggered immediately after the (now) running continuation got reactivated or started.

Triggered during system startup after memory busses and stack have been initialized.

Continuation immediately before the running continuation is deactivated or terminated; to is going to become the next running continuation.

Continuation immediately after the (now) running continuation got reactivated or started.

Agenda

4.1 AOP Mechanisms Under the Hood
4.2 Study: i4Weathermon AOP
4.3 CIAO
4.4 CIAO Results
4.5 Summary
4.6 References
### Kernel Latency Comparison with “OSEK”

#### Table 5. CiAO-AS kernel concern implemented as aspects with number of affected AUTOSAR-OS Features Implemented

<table>
<thead>
<tr>
<th>Concern</th>
<th>CiAO</th>
<th>OSEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plug &amp; Play of optional features and policies!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple activation support</td>
<td></td>
<td>3 task</td>
</tr>
<tr>
<td>Full preemption</td>
<td>1 2 6</td>
<td>3 points of rescheduling</td>
</tr>
<tr>
<td>OS application support</td>
<td>1 2 3 scheduler, task, ISR</td>
<td></td>
</tr>
<tr>
<td>Alarm support</td>
<td>1</td>
<td>1 API</td>
</tr>
<tr>
<td>Resource tracking</td>
<td>1 3</td>
<td>4 task, ISR</td>
</tr>
<tr>
<td>Event support</td>
<td>1 5</td>
<td>5 scheduler, API, task, alarm</td>
</tr>
<tr>
<td>Alarm support</td>
<td>1</td>
<td>1 API</td>
</tr>
<tr>
<td>Full preemption</td>
<td>1 2</td>
<td>6</td>
</tr>
<tr>
<td>Mixed preemption</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context check</td>
<td>1 1</td>
<td>x</td>
</tr>
<tr>
<td>Disabled interrupts check</td>
<td>1 1</td>
<td>30</td>
</tr>
<tr>
<td>Enable w/o disable check</td>
<td>1 3</td>
<td>3</td>
</tr>
<tr>
<td>Missing task end check</td>
<td>1 1</td>
<td>1</td>
</tr>
<tr>
<td>Out of range check</td>
<td>1 1</td>
<td>4</td>
</tr>
<tr>
<td>Invalid object check</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>Error hook</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Protection hook</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Startup / shutdown hook</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pre-task / post-task hook</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Enable w/o disable check</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Disabled interrupts check</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Mixed preemption</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Full preemption</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**CiAO Results**

| (a) voluntary task switch | 160 | 178 | 218 |
| (b) forced task switch | 108 | 127 | 280 |
| (c) preemptive task switch | 192 | 219 | 274 |
| (d) system startup | 194 | 194 | 399 |
| (e) resource acquisition | 19 | 56 | 54 |
| (f) resource release | 14 | 52 | 41 |
| (g) resource release with preemption | 240 | 326 | 294 |
| (h) category 2 ISR latency | 47 | 47 | 47 |
| (i) event blocking with task switch | 141 | 172 | 224 |
| (j) event setting with preemption | 194 | 232 | 201 |
| (k) comprehensive application | 748 | 748 | 1216 |

**Execution time [clock ticks] on TC1796@50 MHz**

<table>
<thead>
<tr>
<th>(ac++1.0pre3 with tricore-g++3.4.3 -O3 -fno-rtti -funit-at-a-time -ffunction-sections -Xlinker --gc-sections)</th>
</tr>
</thead>
<tbody>
<tr>
<td>748</td>
</tr>
<tr>
<td>748</td>
</tr>
<tr>
<td>1216</td>
</tr>
</tbody>
</table>

**CiAO outperforms the marked leader in 11 out of 12 cases by up to 260 percent.**

**CiAO outperforms the marked leader in 11 out of 12 cases by up to 260 percent.**

**CiAO achieves excellent granularity!**
Discussion: Aspect-Aware Development

- By AAD CiAO achieves excellent properties [3–5]
  - configurability and granularity even for fundamental kernel policies
  - complete separation of concerns in the implementation

- The approach has also been applied to other system software
  - PUMA, the C/C++ transformation framework behind ac++ [9]
  - CiAOP, an aspect-oriented IP stack for embedded systems [2]

- Issues: comprehensibility & tool support
  - CiAO: aspect code/base code = 1/2.4
    - where the heck xyz is implemented?
  - calls for additional tool support
  - ac++ weaver implementation is stable, but not as mature as gcc
    - missing or confusing error messages
    - no support for weaving in template code
    - no C-0x support

Summary

- Aspect-Aware Development exploits AOP mechanisms to achieve separation of concerns in configurable system software
  - Advice inverses the direction in which control-flow relationships are established: C calls A ⇒ A advises C
    - advice-based binding as a means to integrate (optional) features
  - Pointcuts provide for an implicit quantification of this integration
    - advice-based binding as a means to integrate interacting features

- CiAO applies these concepts from the very beginning
  - loose coupling by advice-based binding
  - visible transitions by explicit join points
  - minimal extensions by extension slices

- The results are compelling
  - configurability of even all fundamental system policies
  - excellent granularity and footprint

Agenda

4.1 AOP Mechanisms Under the Hood
4.2 Study: i4Weathermon AOP
4.3 CiAO
4.4 CiAO Results
4.5 Summary
4.6 References

Referenzen


This is research, after all :-(


