# 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 12 – 2012-06-06

http://www4.informatik.uni-erlangen.de/Lehre/SS12/V_KSS

### Implementation Techniques: Classification

<table>
<thead>
<tr>
<th>Decompositional Approaches</th>
<th>Compositional Approaches</th>
<th>Generative Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components → Configuration → Variant</td>
<td>Components → Configuration → Variant</td>
<td>Components → Configuration → Generator → Variant</td>
</tr>
</tbody>
</table>

- Text-based filtering (untyped)  
- Preprocessors
- Language-based composition mechanisms (typed)  
- OOP, AOP, Templates
- 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

---

**Notation**

<table>
<thead>
<tr>
<th>BaseClass</th>
<th>method()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>explicitJP()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SomeClass</th>
<th>protected()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>static()</td>
</tr>
<tr>
<td></td>
<td>inadvertable()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AnAspect</th>
<th>exec(&quot;explicitJP&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>intro(&quot;SomeClass&quot;)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASlice</th>
<th>state_variable_</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>anotherMethod()</td>
</tr>
</tbody>
</table>

**AOP Mechanisms Demystified: "Obliviousness"**

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

- With OOP:
  - $PL$ has to call $F_1 \sim PL$ 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 \~ 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
  \( \leadsto \) Aspects can be kept oblivious of the surrounding program
- 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**

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

**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()` into Sink
- Gives advice to `Sink::process()` to invoke `process_data()`
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 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

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 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

Solution Space

The C++ AOP Approach | 4.2 Study: i4WeatherMon AOP

Weather

Sink

Display
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 */ };
advice execution("void Sink::process(const Weather&)")
: after () { ... /* transmit record */ };

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

AOP is as efficient as CPP-based configurability!

I4WeaterMon (AOP): Evaluation

General
1. Separation of concerns (SoC) ✔
2. Resource thriftiness ✔

Operational
3. Granularity
   - Every component implements functionality of a single feature only.
4. Economy
   - All control-flow bindings are established at compile time.
5. Pluggability
   - Sensors and actuators integrate themselves by aspects.
6. Extensibility
   - "Plug & Play" of sensor and actuator implementations.
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 [9, 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
  - using aspects as the primary composition technique

CiAO: General Structure

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

Layers ↦ 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
  ~ "visible transitions"

// yields all hardware invocations from the system layer
pointcut OSToHW() = call("% hw::...::%(...)") && within("% os::...::%({})");

Methodology: Principles of Aspect-Aware Development

Design Principles ↦→ Development Idioms

1. loose coupling by advice-based binding
2. visible transitions by explicit join points
3. minimal extensions by 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!

Methodology: Roles of Aspects and Classes (Cont’d)

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
Example: Mechanism Integration

- **Serial0**
  - `init()`
  - `sendBlock()`

- **Sched**
  - `init()`
  - `ready()`

- **Serial0Ext**
  - `exec("init")`

- **Serial0_Init**
  - `exec("init")`

- **Sched_Init**
  - `exec("init")`

- **Serial0Ext_Task**

  - Task_task
    - `exec("init")`

- **extension aspect**
  - `Serial0Ext { ... }`

- **binding aspect**
  - `Serial0_Init { ... }`
  - `Sched_Init { ... }`

- **slice**
  - `Serial0Ext_Task { ... }`

- **os::krn hw::hal**

- **visible transitions**
  -`init()`

- **This might lead to additional functional dependencies**

---

Example: Policy Integration

- **Cooperative Scheduling**
  - `reschedule()`
  - `setNeedReschedule()`
  - `leaveKernel()`
  - `enterKernel()`

- **Preemptive Scheduling**
  - `reschedule()`
  - `setNeedReschedule()`
  - `leaveKernel()`
  - `enterKernel()`

- **visible transitions**
  - by explicit join points

- **This might lead to additional functional dependencies**

---
Example: Policy Integration

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

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

Methodology: Explicit Join Points

- Advice-based binding \(\rightarrow\) availability of the “right” join points
  - for all semantically important transitions in the system
  - statically evaluable
- Fine-grained component structure \(\sim\) many implicit join points, but
  - amount and precise semantics often implementation dependent
  - aspects have to “know” \(\sim\) no obliviousness
- Important transitions not available for technical reasons as JPs
  - target code may be fragile (e.g., context switch) \(\sim\) must not be advised
  - target code may be written in assembly \(\sim\) transitions not visible as JPs
- Transition join points (T) represent semantically important control-flow transitions inside the kernel
  - level transitions: `user \(\rightarrow\) kernel, user \(\rightarrow\) interrupt`
  - context transitions: `threadA \(\rightarrow\) threadB`
Example: Optional Feature Interaction

ResourceSupport_Task

Priority

OriginalPriority

Occupied

ReleaseResource

GetResource

MixingPreemption_Task

FullPreemption

TaskFunction

Scheduler

ThreadFunc

CleanupSupport

ActiveSupport

ResourceSupport_PCP

ResourceSupport_Sched

MixedPreemption_Task

Preemptable

FullPreemption

Agenda

4.1 AOP Mechanisms Under the Hood
4.2 Study: i4Weathermon AOP
4.3 CAO
4.4 CAO Results
4.5 Summary
4.6 References
### Feature Granularity of CiAO

(static memory demands per feature)

<table>
<thead>
<tr>
<th>concern</th>
<th>CiAO</th>
<th>OSEK</th>
<th>Extension of advice-based binding to</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISR cat. 1 support</td>
<td>1</td>
<td>1</td>
<td>2 + n API, OS control, ISR bindings</td>
</tr>
<tr>
<td>ISR cat. 2 support</td>
<td>1</td>
<td>1</td>
<td>3 + n API, OS control, scheduler, ISR bindings</td>
</tr>
<tr>
<td>ISR abortion support</td>
<td>1</td>
<td>2</td>
<td>1 + m API, OS control, scheduler, n ISR functions</td>
</tr>
<tr>
<td>Resource support</td>
<td>1</td>
<td>3</td>
<td>5 scheduler, API, task, PCP policy implementation</td>
</tr>
<tr>
<td>Resource tracking</td>
<td>1</td>
<td>3</td>
<td>4 task, ISR, monitoring of Get/ReleaseResource</td>
</tr>
<tr>
<td>Event support</td>
<td>1</td>
<td>5</td>
<td>5 scheduler, API, task, alarm, trigger action JP</td>
</tr>
<tr>
<td>Alarm support</td>
<td>1</td>
<td>1</td>
<td>1 API</td>
</tr>
<tr>
<td>OS application support</td>
<td>1</td>
<td>2</td>
<td>3 scheduler, task, ISR</td>
</tr>
<tr>
<td>Full-preemption</td>
<td>1</td>
<td>2</td>
<td>6 3 points of rescheduling</td>
</tr>
</tbody>
</table>

Mixed preemption.

Multiple aspects:
- Stack monitoring
- Context check
- Disabled interrupts check
- Enable w/o disable check
- Missing task and check
- Out of range check
- Invalid object check
- Error hook
- Protection hook
- Startup / shutdown hook
- Pre-task / post-task hook

Plug & Play of optional features and policies!

CiAO achieves excellent granularity!

### Kernel Latency Comparison with “OSEK”

<table>
<thead>
<tr>
<th>test scenario</th>
<th>CiAO</th>
<th>OSEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) voluntary task switch</td>
<td>160</td>
<td>178</td>
</tr>
<tr>
<td>(b) forced task switch</td>
<td>108</td>
<td>127</td>
</tr>
<tr>
<td>(c) preemptive task switch</td>
<td>192</td>
<td>219</td>
</tr>
<tr>
<td>(d) system startup</td>
<td>194</td>
<td>194</td>
</tr>
<tr>
<td>(e) resource acquisition</td>
<td>19</td>
<td>56</td>
</tr>
<tr>
<td>(f) resource release</td>
<td>14</td>
<td>52</td>
</tr>
<tr>
<td>(g) resource release with preemption</td>
<td>240</td>
<td>326</td>
</tr>
<tr>
<td>(h) category 2 ISR latency</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>(i) event blocking with task switch</td>
<td>141</td>
<td>172</td>
</tr>
<tr>
<td>(j) event setting with preemption</td>
<td>194</td>
<td>232</td>
</tr>
<tr>
<td>(k) comprehensive application</td>
<td>748</td>
<td>748</td>
</tr>
</tbody>
</table>

Execution time [clock ticks] on TC1796@50 MHz

```
(ac++1.0pre3 with tricore-g++3.4.3 -O3 -fno-rtti -funit-at-a-time -ffunction-sections
-Xlinker --gc-sections)
```

CiAO outperforms the marked leader in 11 out of 12 cases by up to 260 percent.
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++ [10]
- CiAO IP, 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

<table>
<thead>
<tr>
<th></th>
<th>Base code</th>
<th>Aspect code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files LOC</td>
<td>423 27,086</td>
<td>333 5,923</td>
</tr>
<tr>
<td>LOC</td>
<td>553 5,923</td>
<td>333 5,923</td>
</tr>
</tbody>
</table>

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 \(\Rightarrow\) 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

Referenzen


