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 13 – 2013-05-23

http://www4.informatik.uni-erlangen.de/Lehre/SS13/V_KSS

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

---

### Notation

- **BaseClass**
  - `method()`
  - `explicitJP()`

- **SomeClass**
  - `protected()`
  - `static()`
  - `inadvisable()`

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

- **ASlice**

- **"slice" ASlice**
  - `state_variable_`
  - `anotherMethod()`

---

### AOP Mechanisms Demystified: "Obliviousness"

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

**With OOP:**
- $PL$ has to **call** $F_1 \rightsquigarrow$ $PL$ has to **know** $F_1$.
- Control flows can only be established in the direction of knowledge.

**With AOP:**
- $F_1$ can give **advice** to $PL \rightsquigarrow$ $F_1$ has to **may know** $PL$.
- Control flow is established **opposite** to the direction of knowledge.
- Binding is inherently loose $\rightsquigarrow$ silently missed, if $PL$ does not exist.
Szenario: (Nonfunctional) feature component \( F_1 \) shall be integrated into (optional) SPL components \( \text{PLC}_{1...n} \).

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

Advice inverses the direction in which control-flow relationships are established: \( C \text{ calls } A \implies A \text{ advises } C \)

- 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 \( 0...M...n \) 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

I4WeatherMon: Feature Model

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

How to achieve \( \text{Granularity, Economy, Pluggability, Extensibility?} \)

- 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

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!

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(Temperature const & temp) {}
  };
  advice execution("void Sink::process()") : after() {
    theSink.process_data( _theWeather._pressure );
  }
};

I4WeatherMon: AOP Generic Actuator Integration

aspect Display {
// display each element of the weather data
  advice execution("void Sink::process_data(Wind)") : before() {
    typedef JoinPoint<
      template<
        typename Data,
        typename tjp = arg<0>::ReflectedType Data
      > tjp;
    tjp->arg<0>()->str_val(val);
    print( Data::name(), val, Data::unit() );
  }
};

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
I4WeaterMon: AOP Generic Actuator Integration

```java
aspect SNGConnection : public PCConnection {
    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&)")
                : before () { /* 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)
                : before () { /* transmit record */ }
                : p = pressure._p - 850;
            ;

        advice execution("void Sink::process_data(...)") && args (temp)
            : before (const Temperature &temp)
                : _t1 = (UInt8)temp._t1;
                : _t2 = temp._t2;
            ;
    }
};
```

AOP vs. CPP – Footprint

AOP is as efficient as CPP-based configurability!

I4WeaterMon (AOP): Evaluation

- 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
  - Explicit Join Points
  - 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 [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::...::%(...)");
```

CiAO ↦→ CiAO is Aspect-Oriented

A product line of aspect-oriented operating systems
- Implements OSEK VDX / AUTOSAR OS [1, 8]
- Fine-grained configurability of all system policies and abstractions
- Developed from scratch with AOP

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

Serial0
init()
sendBlock()

Sched
init()
ready()

Serial0Ext
exec("init")

Serial0_Init
exec("init")

Sched_Init
exec("init")

Visible transitions by explicit join points

os::krn
hw::hal

binding aspects

extension aspects

upcall aspects
Example: Mechanism Integration

- binding aspects implement upcalls by advice-based bindings

Example: Mechanism Integration

- extension aspects add to mechanisms by extension slices

Example: Policy Integration

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

- policy aspect Sched_ASTBinding
  "setNeedReschedule"
  "leaveKernel"
  "enterKernel"

- policy aspect Kernel_ASTSync
  "enterKernel()"
  "leaveKernel()"

Cooperative Scheduling:

Preemptive Scheduling:

visible transitions by explicit join points

Additional functional dependencies

This might lead to additional functional dependencies
Example: Policy Integration

Cooperative Scheduling:

- `enterKernel`
- `leaveKernel`

Preemptive Scheduling:

- `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

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 intitialization or entering of the idle state
- 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

Type representing function or method

- `internalErrorHook()`
- `internalProtectionHook(statusType error)`
- `internalPreTaskHook()`
- `internalPostTaskHook()`
- `internalStartHook()`
- `internalStopHook()`

**Description**

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.

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

**Entry point of a new thread (continuation).**

- `ThreadFunc()`

**Entry point of the respective AST.**

- `exec ThreadFunc()`

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

...
Autosar-OS Features Implemented in CiAO as Aspects (Excerpt)

Kernel Latency Comparison with "OSEK" [4]

<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>

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

Execution time [clock ticks] on TC1796@50 MHz
(ac++1.0pre3 with tricore-g+++3.4.3 -O3 -fno-rtti -funlit-at-a-time -ffunction-sections-Xlinker --gc-sections)

Kernel Latency Comparison with "OSEK" [4]

<table>
<thead>
<tr>
<th>feature with feature or instance</th>
<th>text</th>
<th>data</th>
<th>bss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base system (OS control and tasks)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>per task</td>
<td>+func</td>
<td>+20</td>
<td>+16+stack</td>
</tr>
<tr>
<td>per application mode</td>
<td>0</td>
<td>+4</td>
<td>0</td>
</tr>
<tr>
<td>per ISR</td>
<td>+func</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per disable-enable</td>
<td>+func</td>
<td>+4</td>
<td>0</td>
</tr>
<tr>
<td>Resource support</td>
<td>128</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per alarm</td>
<td>0</td>
<td>+12</td>
<td>0</td>
</tr>
<tr>
<td>Full preemption</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per join point</td>
<td>+12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mixed preemption</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per join point</td>
<td>+44</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per task</td>
<td>0</td>
<td>+4</td>
<td>0</td>
</tr>
<tr>
<td>Wrong context check</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per void join point</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per StatusType join point</td>
<td>+8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Interrupts disabled check</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per join point</td>
<td>+64</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invalid parameters check</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>per join point</td>
<td>+36</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Error hook</td>
<td>0</td>
<td>+4</td>
<td>0</td>
</tr>
<tr>
<td>per join point</td>
<td>+54</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Startup hook or shutdown hook</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pre-task hook or post-task hook</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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++ [10]
- CiAOIP, 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>Base code</th>
<th>Aspect code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files LOC</td>
<td>Files LOC</td>
</tr>
<tr>
<td>CiAO kernel only</td>
<td>423 21,086 333 5,923</td>
</tr>
<tr>
<td>CiAO COM</td>
<td>112 8,689 297 5,552</td>
</tr>
<tr>
<td>CiAO IP stack</td>
<td>45 5,038 96 3,230</td>
</tr>
<tr>
<td>CiAO kernel</td>
<td>580 34,813 726 14,705</td>
</tr>
</tbody>
</table>

This is research, after all :-)

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 \(\implies\) A advises C
  where the heck xyz is implemented?
- Pointcuts provide for an implicit quantification of this integration
  where the heck xyz is implemented?
- 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


