Lean And Efficient System Software Product Lines: Where Aspects Beat Objects*

Daniel Lohmann, Olaf Spinczyk, Wolfgang Schröder-Preikschat
{dl, os, wosch}@cs.fau.de

Friedrich-Alexander-University Erlangen-Nuremberg, Germany
Computer Science 4 – Distributed Systems and Operating Systems

Abstract. Software development in the domain of embedded and deeply embedded systems is dominated by cost pressure and extremely limited hardware resources. As a result, modern concepts for separation of concerns and software reuse are widely ignored, as developers worry about the thereby induced memory and performance overhead. Especially object-oriented programming (OOP) is still little in demand. For the development of highly configurable fine-grained system software product lines, however, separation of concerns (SoC) is a crucial property. As the overhead of object-orientation is not acceptable in this domain, we propose aspect-oriented programming (AOP) as an alternative. Compared to OOP, AOP makes it possible to reach similar or even better separation of concerns with significantly smaller memory footprints. In a case study for an embedded system product line the memory costs for SoC could be reduced from 148–236% to 2– 10% by using AOP *instead* of OOP.

1 Introduction

The domain of embedded and deeply embedded devices is dominated by 8 bit μ-controllers with 0.25 - 4 KB of RAM and a few KB of flash memory. In 2000, more than 98% of the total worldwide CPU production (8 billion entities) were dedicated to the domain of embedded systems; 87% of the entities used in this domain were 8 bit or smaller[47]. From the viewpoint of procurement, this "old-fashioned technology" is still the best compromise with respect to functionality and costs. In areas of mass production a *few cents* decide over market success or failure – a situation that can not be expected to change soon, given that the envisioned scenarios of *smart dust*[30], *ubiquitous computing*[48] and *proactive computing*[47]) crucially depend on the bulk availability of very cheap, self-organizing "intelligent" devices.

Counting cents in hardware procurement basically leads to counting bytes in software development. To cope with hardware cost pressure and extremely limited resources, software developers for deeply embedded systems intentionally avoid modern language concepts for a better separation of concerns (SoC), as they worry about the thereby induced memory and performance overhead. Especially object-oriented programming (OOP) is still little in demand, as some of its fundamental concepts (e.g. *late*

* This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4 and SP 968/2-1.

FINAL DRAFT, to appear in: Transactions on Aspect-Oriented Software Development (TAOSD)
Special Issue on Systems & Middleware, Springer LNCS

binding by virtual functions) are known to have non-negligible costs[20]. Hence, most software development for embedded systems is still performed "ad-hoc" in C (often even assembler) with a strong focus on minimizing hardware requirements. Organized reuse and separation of concerns is often considered as less important.

System Software Product Lines

The common "ad-hoc" application-specific development approach is doomed to fail for reusable system software, such as operating systems or light-weight middleware for embedded devices. System software for this domain has not only to cope with the above mentioned resource constraints, but also with a very broad variety of functional and non-functional requirements[10]. It has to be tailorable to provide exactly the functionality required by the intended application, but nothing more. This leads to a *family-based* or *product line* approach, where the variability and commonality among family members is expressed by *feature models*[16]. The tailorability of software product lines depends mostly on the offered level of functional exchangeability (here denoted as *variability*) and functional selectability (here denoted as *granularity*). Both, variability and granularity, require a (right-unique) mapping from implementation components to the features they implement, thus, a good separation of concerns. As the principal overhead of OOP is not acceptable in this domain, we advocate to use *aspect-oriented programming (AOP)* instead.

AOP, as well as OOP, provides means for a better separation of concerns. Today, AOP is mostly perceived as an *extension* to OOP, which leads to the impression that it has to induce similar or even higher overhead. This is understandable, as most aspect languages are actually extensions to object-oriented languages like Java. For AspectJ[31], studies show furthermore that AOP concepts may indeed lead to some extra overhead [21]. Nevertheless, AOP itself is not limited to OOP. Language extensions have as well been proposed for non-OOP languages (such as AspectC[12]) and multi-paradigm languages (such as AspectC++[46]). In these languages it is possible to use AOP not only as extension, but also as *alternative* to OOP.

Objectives

This article shows that for the specific requirements of resource-constrained software development, AOP can be superior to OOP. Our focus is *not* on separation of crosscutting concerns and other structural benefits of AOP that already have been discussed in many papers. Instead, we want to draw attention to a mostly unexplored benefit of AOP: Given a well-optimizing static weaver, AOP can provide SoC more *resource-efficiently* than OOP. Our goals are in particular to:

- broaden the perception of AOP. Currently AOP is mostly perceived as an extension to OOP for a better SoC. Understanding it as an *alternative* to OOP with respect to *resource-efficiency* and *hardware costs* is a new point of view.
- attract new communities to AOP. Especially the systems and embedded communities have a profound skepticism regarding the suitability of high-level programming paradigms ("everything beyond C"), as they (have to) favor efficiency over SoC.

Demonstrating that with aspects it can be possible to reach SoC without having to give up efficency would help to broaden the acceptance of AOP.

For this purposes, we conducted a study in which we compare the resource requirements of three implementations of the same embedded software product line. A highly efficient C-based implementation that does not provide SoC, an object-oriented, and an aspect-oriented implementation. We evaluate the OO and AO implementations regarding the techniques used to reach SoC. The focus is on the thereby induced *costs*. While both implementations reach the SoC goals, the OO-based solution leads to significantly higher resource requirements than the AO-based solution, which can compete with plain C.

Structure of the Paper

The paper is organized as follows: In the next section, we present our case study about an aspect-oriented and an object-oriented implementation of an embedded software product line. Both implementations are analyzed with respect to used idioms and potential cost drivers. In section 4, these implementations are further investigated regarding their resource requirements. The results are discussed in section 5. Section 6 provides an overview of related work. Finally, the paper is briefly summarized in section 7.

2 Scenario

In the following, we present and analyze a case study of an embedded software product line with three different implementations: C-based, OOP-based, and AOP-based. For the sake of comprehensibility, we have conducted this case study in a more application-oriented field. The scenario is an *embedded weather station* program family that should be configurable in various different variants. It is a somewhat typical example of an embedded application software product line. The identified patterns and results are, however, equally relevant for the development of system software, which is our main field of research.

2.1 Overview

A weather station variant basically consists of one or more *sensors* to gain environmental information and one or more *actors* to process the gathered weather data. Figure 1 shows the possible variants of the weather station as a feature model. Note that the list of available sensors as well as actors is expected to grow in future versions, e.g. by additional sensors for wind direction and humidity. Not visible in the feature model, but nevertheless part of the product line definition, is that we have to distinguish between two kinds of actors:

generic actors are able to process / aggregate information of any set of sensors. *Display* and *XMLProto* are examples for generic actors. The XML representation of weather information, for instance, can easily and automatically be extended for additional sensors.

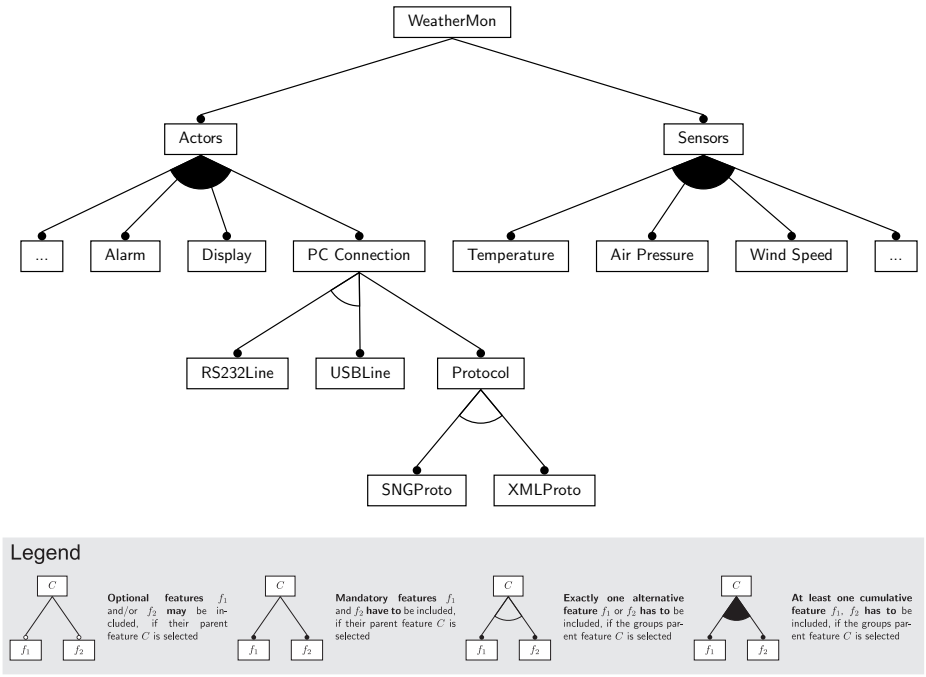

Fig. 1. Feature diagram of the embedded weather station product line. A *WeatherMon* variant consists of at least one sensor and one actor. *Sensors* gain information about the environment, such as *Temperature*, *Air Pressure*, or *Wind Speed*. *Actors* process sensor data: Weather information can be printed on a (LCD-) *Display*, monitored to raise an *Alarm* if values exceed some threshold, and passed to a PC over a *PCConnection*, which can be either an *RS232Line* or an *USBLine*, using either an XML-based data representation (*XMLProto*) or a proprietary format (*SNGProto*).

non-generic actors process / aggregate sensor data of some specific sensors only. *SNGProto* is an example for a specific actor. It implements a legacy binary data representation for compatibility with existing PC applications. This format encodes information of certain sensors only.

2.2 Hardware Platform

On the hardware side, sensors and actors of the weather station are connected to a small μ-controller (Figure 2). The AVR series by Atmel is a typical μ-controller product line. It is based on an 8 bit RISC core and offered with a broad variety of on-board equipment: 0-4 KB RAM, 1-128 KB program memory (flash), various busses and connectors (I^2C, serial lines, A/D converters). Wholesale prices of the chip scale between 0.3 EUR and 7 EUR. They depend mostly on the on-board equipment, especially the amount of RAM and flash memory.

The goal is to reach a similar level of scalability in the software product line, so that hardware costs do scale with the amount of selected features.

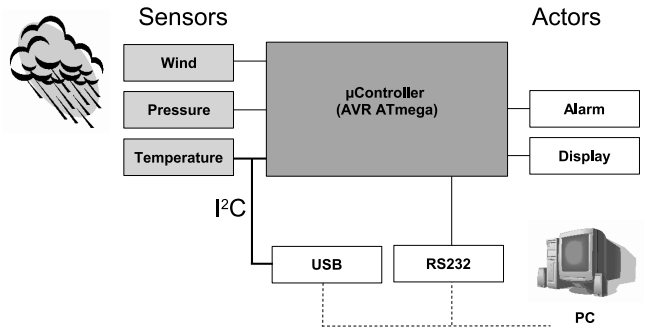

Fig. 2. Weather station hardware platform

2.3 Configuration Process

A weather station variant is configured by selecting features from the feature model. Conceptually, the feature model can be considered as the description of the *configuration space*, while a (valid) feature selection is a description of a concrete variant. In the *implementation space*, a *family model* provides a mapping from the abstract features to concrete implementation artifacts (such as .h and .cpp files) which implement the selected features. The feature selection is evaluated against the configurations space by a variant management tool[9], which copies the thereby defined set of implementation artifacts from the component repository into an output directory. The build process is then started in this directory to compile and link the configured system into the concrete variant (Figure 3).

The approach to map conceptual features by a family or platform model to concrete implementation artifacts is quite common the domain of embedded system software such as operating system product lines.[1] The main advantages are flexibility and portability. By providing hardware-specific versions of the implementation space (family model and implementation components), the same conceptual OS model and configuration tool can be used to ”generate” highly optimized variants for very different μ-controller platforms and derivates.

The overall configurability (variability and granularity) of such product lines depend, however, on the (right-unique) mapping from implementation components to the features they implement, thus, a good separation of concerns. As a matter of fact, many features can not be mapped to single (C-language) artifacts, their implementation is intermingled within the implementation of other features. The result is scattered and tangled code, typically processed by an intra-artifact “second-order configuration” by means of the C preprocessor and conditional compilation. This hampers maintainability and evolvability of the implementation artifacts and thereby limits on the long term

[1] Examples are eCos[1], PURE[10], or many implementations of the OSEK system software standard used in automotive industry[2].

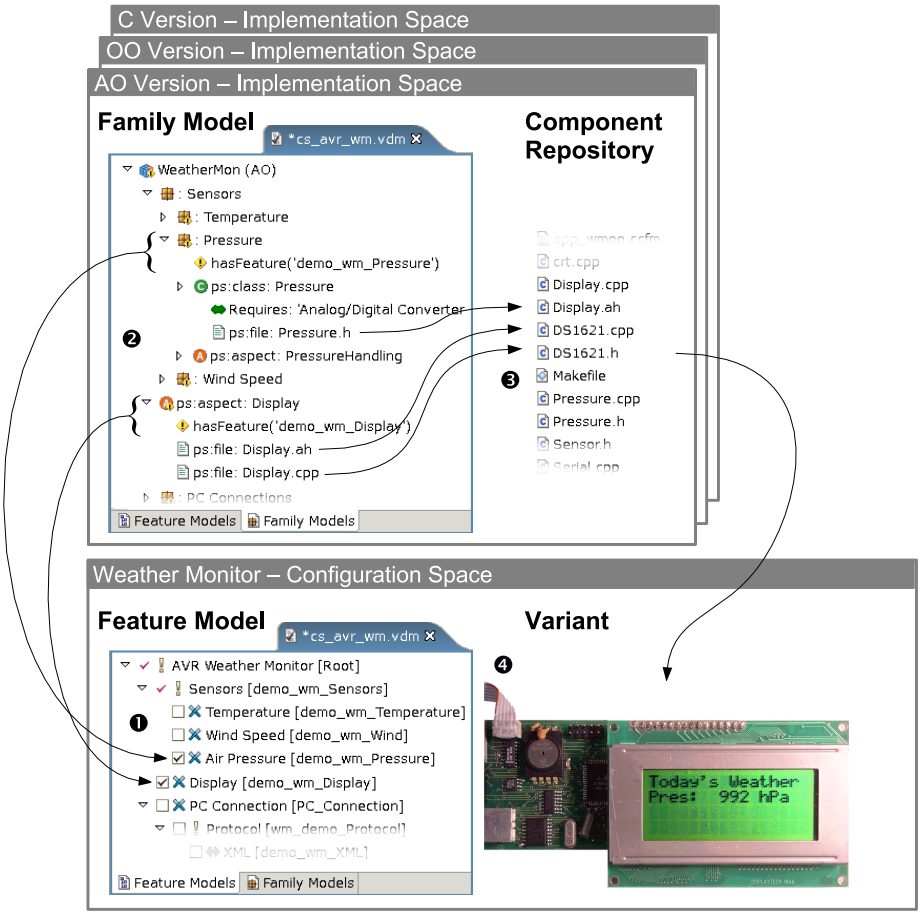

Fig. 3. Weather station configuration process. In the *configuration space* the user selects all wished features from the *feature model* (1). The *implementation space* consists of a *family model* and a *component repository.* The family model maps features to *logical implementation components* (2), which in turn are mapped to *physical implementation files* (3). The thereby determined set of implementation files for a concrete configuration is copied into the target directory, and finally compiled and linked into the actual *weather station variant* (4).

the extensibility of the whole product line. Hence, we strive for a better separation of concerns—as long as it does not lead to significant extra costs on the hardware side.

3 Implementation

In the following, we present and analyze two different implementations of the weather station product line that fulfill the goal of SoC. One implementation is OOP-based, while the other is AOP-based. The implementations were performed by two different,

but equally experienced developers. Both have been (under the premise that SoC has to be achieved) carefully optimized with respect to resource requirements.

A third "traditional" C-based implementation is solely optimized for minimal resource consumption. Its only purpose is to provide the lower bounds of the resource consumption that can be reached in the different product line configurations.

The section is organized as follows: We start with a list of requirements in 3.1, which is followed by a description of the three implementations (C in 3.2, OO in 3.3, and AO in 3.4). These descriptions are intended as a brief overview only. In particular, they do not motivate the chosen design of the OO and AO versions. Instead, all design decisions and idioms used to reach SoC in the OO and AO versions are discussed collectively (to set them in contrast with each other) in 3.5. The results are summarized in 3.6.

3.1 Implementation Requirements

Besides the functional features that have to be implemented, the additional requirements on the implementations can be summarized as *resource-thriftiness* (all versions) and *separation of concerns* (AO and OO versions). This means in particular:

granularity Components should be fine-grained. Each implementation element should be either mandatory (such as the application main loop) or dedicated to a single feature only.

economy The use of expensive language features should be avoided as far as possible. For instance, a method should only be declared as `virtual` if polymorphic behavior of this particular method is required.

pluggability Changing the set of selected sensors and/or actors should not require modifications of any other part of the implementation. This basically means that sensor/actor implementations should be able to integrate themselves into the system.

extensibility The same should hold for new sensor or actor types, which may be available in a future version of the product line.

3.2 Implementation of the C Version

Figure 4 shows the basic structure of the C version. Weather information is updated by the `measure()` function, which invokes – configured by conditional compilation – all existing sensors. For efficiency reasons, the actual weather data is passed as a global variable (not shown). Weather information is processed by the `process()` function, which invokes each actor (such as `display_process()`) – again configured by means of conditional compilation. The `..._process()` function of each actor retrieves weather information for all configured sensors it is interested in – once more using conditional compilation.

Due to the design goal to minimize the resource consumption, the C version does *not* fulfill our requirements on SoC (*granularity*, *pluggability*, *extensability*). By using global variables, conditional compilation and inlining of all functions that are referred only once, it offers, however, efficiency.

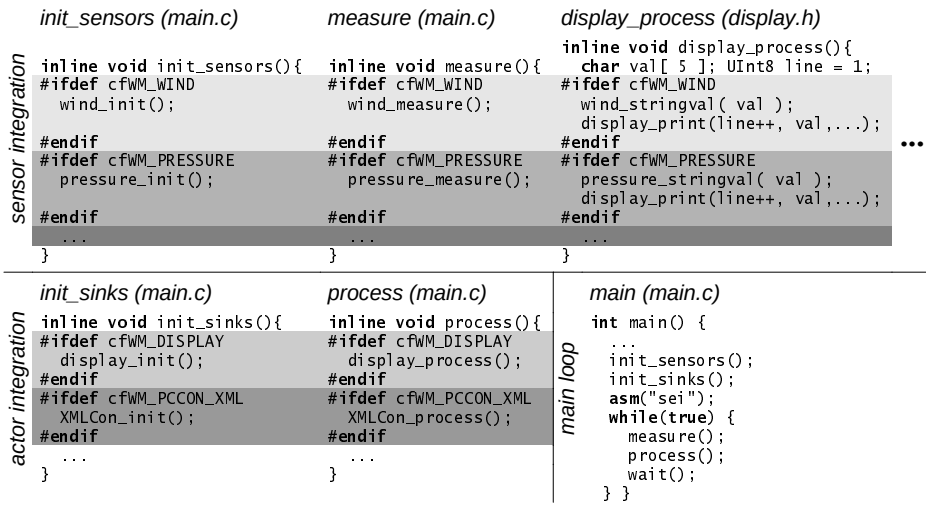

Fig. 4. Static structure of / scattered code in the C version (excerpt)

3.3 Implementation of the OO Version

Figure 5 shows the class model of the OO version. Central elements are the `Weather` and `Sink` classes. `Weather` aggregates all sensors, which are registered at runtime by calling `Weather::registerSensor()`. The `Sink` class aggregates all actors, respectively. Internally both, sensors and actors, are managed by chaining them into light-weight single-linked lists (`ChainBase`).

Principle of Operation

1. Weather information is acquired by calling `Weather::measure()`, which in turn invokes the `Sensor::measure()` method on every registered sensor to update the sensor data.
2. Weather information is processed by the `Sink::process()` method. `Sink::process()` first calls `Actor::before_process()` for each registered actor to initialize the processing.
3. Sensor information is passed to the actors by calling `Actor::process()` for each registered sensor. Actors retrieve the actual sensor name, unit, and measured data (as character strings) by calling the respective Sensor methods.
4. At last, data processing is finalized by `Sink::process()` invoking `Actor::after_process()` on each actor.
5. The whole process is repeated by the application main loop every second.

3.4 Implementation of the AO Version

The class/aspect model of the AO version is shown in Figure 6. Central elements are, again, the classes `Weather` and `Sink`. Each sensor class is accompanied by a *han-*

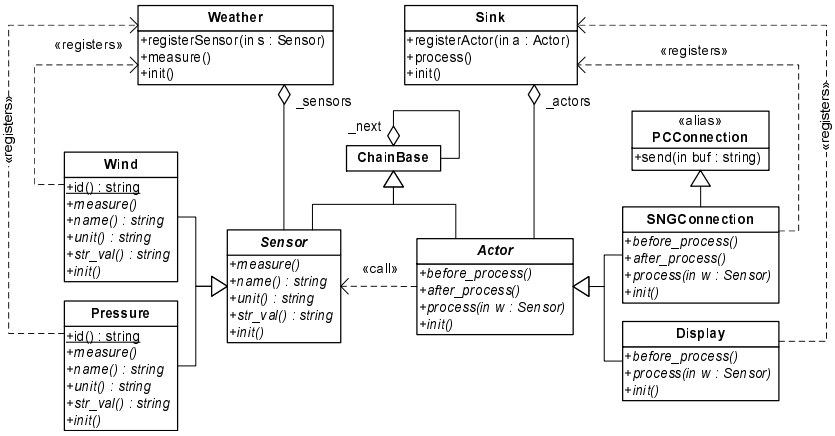

Fig. 5. Static structure of the OO version (excerpt with two sensors/actors, virtual functions are depicted in *italics*, static functions are underlined)

dling aspect, which performs the actual integration into `Weather` and `Sink`. A handling aspect performs two introductions: It aggregates the sensor as an instance variable into class `Weather` and a sensor-specific (empty) `process_data()` method into class `Sink`. Additionally, it defines function execution advice for `Weather::measure()` and `Sink::process`. Actors are implemented as aspects, which define execution advice for `Sink::process()` (for initialization and finalization) and the sensor-introduced `Sink::process_data()` methods (for the actual data processing).

Principle of Operation

1. Weather information is acquired by calling `Weather::measure()`, which is advised by the handling aspects of every sensor to call the sensor's `measure()` method.
2. Weather information is processed by the `Sink::process()` method. `Sink::process()` is before-advised by any actor that needs to initialize before processing.
3. `Sink::process()` is advised for each sensor to call the introduced sensor-specific `process_data()` method.
4. The sensor-specific `process_data()` method is advised by every actor that processes data of this sensor.
5. At last, data processing is finalized by `Sink::process()` being after-advised for any actor that needs to finalize its processing.
6. The whole process is repeated by the application main loop every second.

3.5 Used AOP and OOP Idioms

To achieve the required level of SoC as well as pluggability and extensibility, both versions use approach-specific idioms and patterns in the design and implementation. In

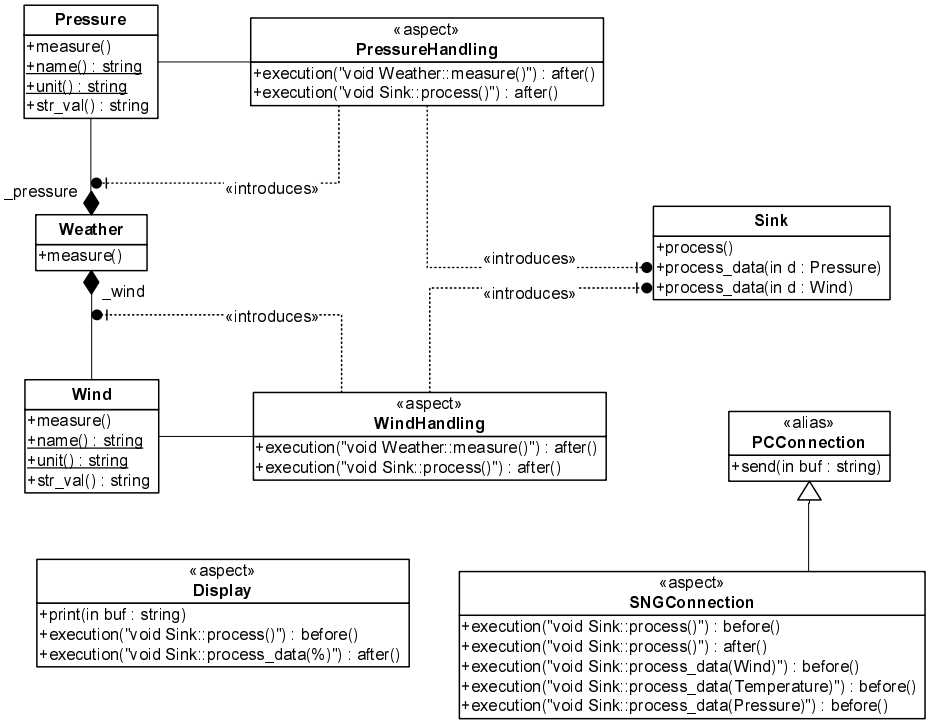

Fig. 6. Static structure of the AOP version (excerpt with two sensors/actors)

the following, we will analyze some interesting parts of the AOP and OOP implementations. The goal is to identify, compare, and discuss the idioms that have to be used to reach SoC. In particular, the following problems had to be solved:

1) Working with configuration-dependent sensors/actors sets
Both implementations have in common that the `Weather`/`Sink` classes are used to abstract from the configured sets of sensors and actors. These two abstractions are unavoidable, because the `main()` function, which performs the endless measurement and processing loop, should be configuration-independent and, thus, robust with respect to extensions. However, the implementation of the 1:n relationship of `Weather`/`Sink` and the concrete sensors and actors is completely different.

Interface dependency between Weather/Sink and sensors/actors
The `Weather`/`Sink` classes need to be able to invoke the measuring/processing of data independent of the actual sensor/actor *types*. Otherwise, pluggability and extensibility would be broken:

OO version: In the OO version, this is solved by *common interfaces* and *late binding*. Sensors have to inherit from the abstract class `Sensor`, actors from the

abstract class `Actor`, respectively. `Weather`/`Sink` invoke sensors/actors via these interfaces, thus depend on them. `Sensor::measure()`, `Actor::before_process()`, `Actor::after_process()`, and `Actor::process()` have to be declared as virtual functions:

```
struct Sensor : public ChainBase {
  virtual void measure() = 0;
  ...
};
...
class Weather {
public:
  ...
  void measure () {
    for( Sensor* s = ... )
      s->measure();       // virtual function call!
  }
};
```

AO version: In the AO version, the interface relationship is reverted. `Weather` and `Sink` do not depend on sensors/actors providing any specific interface. Sensors/actors are integrated by defining function execution advice for `Weather::measure()` and `Actor::process()`:

```
class Weather {
public:
  void measure () {} // empty implementation
};
...
aspect PressureHandling {
public:
  advice "Weather" : Pressure _pressure;
  advice execution ("void Weather::measure()") : before () {
    // non-virtual inlineable function call
    theWeather._pressure.measure ();
  }
  ...
};
```

Potential cost drivers: In the OO version, four methods in two classes have to be declared and called as virtual functions. The AO version induces no costs, as all sensor/actor code can be inlined.

Working with sets of sensors/actors

`Weather`/`Sink` need to be independent of the actual *number* of sensors/actors configured into the system. There is a 1:n relationship between `Weather`/`Sink` and sensors/actors:

OO version: In the OO version, this is solved by a simple *publisher/subscriber* mechanism. Sensors are chained into a linked list of "publishers", which is frequently iterated by `Weather::measure()` to update the data of each sensor:

```
class Weather {
  static  Sensor* _sensors;
public:
  Sensor* firstSensor() const {
    return _sensors;
  }
  void measure () {
    for( Sensor* s = firstSensor(); s != 0;
         s = static_cast< Sensor* >( s->getNext() ) )
      s->measure();
```

```
    }
    ...
};
```

Actors are similarly chained into a list of "subscribers" which are invoked if new sensor data is available. The `Sink` class acts as "mediator" between sensors and actors. For the sake of memory efficiency, actors do not subscribe for single sensors, but are implicitly subscribed for the complete list.

AO version: In the AO version, actors and sensors are implicitly chained at compile-time by multiple aspects defining advice for the same `Weather::measure()` and `Sink::process()` execution join points.

Potential cost drivers: In the OO version, `Weather`/`Sink` as well as sensors/actors need to carry an extra pointer for the chaining. Some additional code is required to iterate over the chain of sensors/actors.[2]

Registration of sensors/actors

This problem is closely related to the previous one. Sensors and actors need to be able to register themselves for the publisher/subscriber chain:

OO version: In the OO version, registration is done at runtime by calling `Weather::registerSensor()`/`Sink::registerActor()`. Self-registration requires some C++ trickery: Each sensor/actor is instantiated as global object. The registration is performed by the constructor, which is "automatically" triggered during system startup.

AO version: In the AO version, no extra efforts are required for self-registration. The chaining of actors and sensors is implicitly performed at compile time by advice-code weaving. Basically, it is the presence of some aspect in the source tree that triggers the registration.

Potential cost drivers: In the OO version, the use of global object instances causes some overhead, as the compiler has to generate extra initialization code and puts a reference to this code in a special linker section. A system startup function has to be provided that iterates over this section and invokes the constructors of all global instances.

2) Implementation of generic actors

Generic actors process sensor data from any sensor type. For example, the Display actor should print the measured values of any configured sensor, regardless of the current configuration or future extensions with new sensor types. However, this leads to an interface dependency between actors and sensors, because (at least) the value has to be obtained in a generic way:

---

[2] The `firstSensor()` and `getNext()` operations are, however, inlined as they perform just a pointer lookup.

OO version: In the OO version, this is again solved by *interfaces* and *late binding*. To enable actors to retrieve sensor information from any sensor, the `Sensor` interface has to be extended by three additional virtual functions: `Sensor::name()`, `Sensor::unit()`, and `Sensor::str_val()`:

```
class Display : public Actor {
public:
    UInt8 _line;
  // print a line on the display, increment _line
  void print(const char *name, const char *val_str, const char *unit );
  ...
  virtual void before_process() {
    _line = 1;
  }
  // called by Sink::process for every sensor
  virtual void process( Sensor* s ) {
    char val[ 5 ];
    s->str_val( val );
    print( s->name(), val, s->unit() );
  }
};
```

AO version: In the AO version, this is solved by using the AspectC++ concept of *generic advice* [32, 35]. For every sensor `MySensor`, the corresponding `MySensorHandling` aspect introduces an empty sensor-specific `process_data(const MySensor&)` method into class `Sink` and gives advice to `Sink::process()` to invoke the introduced method:

```
aspect PressureHandling {
public:
  advice "Weather" : Pressure _pressure;
  ...
  // introduce an empty process_data function for the pressure
  advice "Sink" : void process_data( const Pressure & ) {}
  // call Sink::process_data for the pressure
  advice execution("void Sink::process()") : after () {
    theSink.process_data( theWeather._pressure );
  }
};
```

An actor gives generic advice that matches every (overloaded) `Sink::process_data()` function. To be independent of the actual sensor type, the advice body uses the AspectC++ join point API to retrieve a typed reference to the sensor instance:

```
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[5];
    tjp->arg<0>()->str_val( val );
    print( Data::name(), val, Data::unit() );
  }
};
```

AspectC++ instantiates advice bodies per join point. Therefore, the calls to the actual sensor's `str_val()`, `name()`, and `unit()` methods can be bound at compile-time. As an additional optimization, `name()` and `unit()` are implemented as static (class) functions.

Potential cost drivers: In the OO version, three additional virtual functions are required in the sensor classes, as well as additional virtual function calls in the actor classes. In

the AO version, the join point API has to be used for a uniform and type-safe access to the sensor instance, which induces some overhead. Furthermore, the advice body of a generic actor is instantiated once per sensor, which may lead to code bloating effects.

3) Implementation of non-generic actors
Non-generic actors process data of some sensors only. The legacy SNG protocol, for instance, encodes weather data in a record of wind speed, temperature and air pressure. It exposes the actual data using sensor-specific interfaces. The record may be sparse, meaning that a specific sensor, such as Temperature, may or may not be present in the actual system configuration.

OO version: In the OO version, a non-generic actor filters the sensors it is interested in by *runtime type checks* in the `process()` method. Each passed sensor is tested against the handled sensor types. If the runtime type matches with a handled sensor, a downcast is performed to get access to the sensor-specific interface:

```
class Pressure : public Sensor {
  static const char *_id;
public:
  ...
  static const char * id () { return _id; }
  const char *name () const { return _id; }
};
...
class SNGConnection : public Actor, protected PCConnection {
    UInt8 _p, _w, _t1, _t2;     // weather record
public:
  virtual void before_process() { ... /* init record */ };
  virtual void after_process() { ... /* transmit record */ };
  ...
 // collect wind, pressure, temperature data
  virtual void process( Sensor* s ) {
  const char *name = s->name();
    if (name == Wind::id()) // pointer comparison
      _w = ((Wind*)s)->_w;
    else if (name == Pressure::id())
      _p = ((Pressure*)s)->_p - 850;
    else if (name == Temperature::id()) {
      _t1 = (UInt8)((Temperature*)s)->_t1;
      _t2 = ((Temperature*)s)->_t2;
  } }
};
```

The idiom commonly used for runtime type checks in C++ is the `dynamic_cast` operator, which is part of the C++ *runtime type interface* (*RTTI)*. RTTI is, however, quite expensive, as it requires additional runtime support and leads to some extra overhead in *every* class that contains virtual functions. To avoid this overhead, our implementation uses a "home-grown" dynamic type-check mechanism: The test is performed by comparing the string address returned by the (late-bound) `Sensor::name()` method with the address of the name string stored in the concrete class[3], which is also returned by the static `Sensor::id()` method. The expensive C++ RTTI mechanism has been disabled.

Normally, dynamic type checks are considered harmful, because of a lack of extensibility and the accumulated costs of type checks, which sometimes outweigh the

[3] This basically reduces the overhead of a runtime type test to a virtual function call and a pointer comparison. As the storage for the name string has to be provided anyway, this mechanism also induces no extra overhead in the sensor classes.

costs of a single virtual function call. However, in our case a *non-generic* actor shall be implemented. Therefore, extensibility is not an issue and the overhead of our type check implementation is acceptable. At the same time, alternative designs such a *visitor* [24] fail, because a Visitor interface would have to list a `visitSensor()` method for every sensor type. However, the set of sensors is configurable and should nowhere be hard-wired.

AO version: In the AO version, binding an actor to selected sensors only is realized by giving advice for specific `Sink::process_data()` methods only instead of using generic advice:

```
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");

  advice execution("void Sink::process(const Weather&)")
      : 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;
  }
};
```

Potential cost drivers: Runtime type checks in the OO version induce nevertheless some overhead. In the AO version, some overhead is induced by the `args()` pointcut function, which is used here to get the actual sensor instance.

3.6 Implementation Summary

Both, the OOP as well as the AOP implementation of the embedded weather station product line provide good SoC. In particular, the implementation requirements described in section 3.1 are met by both versions:

granularity is achieved by the OO as well as the AO version. Each implementation component is either mandatory (such as the `Weather` and `Sink` classes), or dedicated to a single feature only.

economy is achieved as far as possible. In the OO version, only methods that have to be available via a generic interface are declared as virtual. RTTI is not used, as the required runtime type checks can be implemented with less overhead. In the AO version, join point-specific context information is used only sparingly.

pluggability is achieved as well. In both versions, no component has to be adapted if the set of selected sensors/actors is changed. Sensors and actors basically integrate themselves, if their implementation component is present in the configured source tree. The OO version uses global instance construction for this purpose. In the AO version, the integration is performed by advice.

extensibility is also achieved. In the OO version, new sensor/actor types just need to implement the common `Sensor`/`Actor` interface. In the AO version, new sensor/actor types just need to provide some aspect that performs the integration into `Weather`/`Sink`.

Overall, the AO and OO versions are equipollent from the SoC viewpoint. We identified, however, noticeable more potential cost drivers in the OO version than the AO version. Especially virtual functions were unavoidable in many places to realize loose coupling and genericity of components. In the next section, we analyze how this affects scalability and memory demands of the product line.

4 Cost Analysis

In this section, we analyze scalability and memory requirements of the embedded weather station product line. For this purpose, several configurations of the weather station were generated as AO, OO, and C variants.[4] For each variant, we measured:

- static memory demands, which are determined by the amount of generated machine code (*text*), static initialized data (*data*) and static non-initialized data (*bss*).
- dynamic memory demands, which are determined by the maximum stack space used by the running application (*stack*).[5]
- the runtime of a complete measure/process-cycle.

On the actual hardware, *text* occupies flash memory space. *Data* occupies flash memory *and* RAM space, as it is writable at runtime and therefore has to be copied from flash into RAM during system startup. *Bss* and *stack* occupy RAM space only.

4.1 Measurement Methods

Static memory demands (*text*, *data*, *bss*) could easily be retrieved directly from the linker map file. Dynamic memory demands (*stack*) and runtime had to be measured in the running targets:

[4] All variants were compiled with `avr-g++` (GCC) 3.4.1 using `-Wall -fno-rtti -Os -fno-exceptions -fomit-frame-pointer -ffunction-sections` optimization flags. AO variants were woven with `ac++` 0.9.3.

[5] The weather station software uses no heap, which otherwise would also contribute to dynamic memory demands.

Measuring Stack Utilization
For simple programs it is possible to determine the stack utilization off-line and byte-exact by static analysis of the machine code. Unfortunately in our case the program execution graph is not predictable due to the use of interrupts and late bound functions. Therefore we used runtime monitoring of the stack as a pragmatic alternative. A common technique for runtime stack monitoring is to initialize the entire stack space with some specific *magic pattern* during system startup. The maximum amount of stack used can then be measured at any time by searching (from bottom of stack) for the first byte where the pattern has been overwritten.

In the weather station variants, this technique was used to implement stack measurement as an additional *sensor type*. Understanding stack measurement as just another sensor had some nice advantages. Because of the achieved pluggability (in the AO and OO versions) it was very easy to apply stack measurement to any weather station configuration. Of course, some extra care had to be taken to ensure that the maximum stack utilization is not caused by the stack measurement sensor itself. For this reason, the stack measurement implementation uses only global variables which do not occupy stack space. By analyzing the generated machine code we ensured that the stack utilization of the stack sensor methods is minimal among all sensors. As all sensor methods are invoked from the same call depth level and at least one "real" sensor besides stack measurement is used in a weather station configuration, it can thereby be guaranteed that stack measurement itself does not tamper the maximum stack utilization. In the actual targets, the thereby acquired maximum stack utilization remained stable after a short startup time and could be read from one of the attached generic actors.

Measuring Runtime
Runtime measuring was not implemented as another sensor type, as it had been too difficult to distinguish the runtime taken by the sensor processing itself from the runtime of the target to measure. Instead, we used a less invasive approach that could be implemented with just two additional assembler statements: In the application main loop, a digital I/O port of the AVR μ-controller is set to *high* before the call to `Weather::measure()` and reset to *low* after the return of `Sink::process()`. The result is a rectangular signal on this port, which was recorded and analyzed with a storage oscilloscope[6]. A *high* period in the signal represents the runtime taken by a complete `Weather::measure()` / `Sink::process()` cycle. After a short startup time, period and phase of the signal remained stable and the length of the *high* phase could be measured.

4.2 Overall Scalability of the Product Line

As the graphs in Figure 7 show, the resulting RAM/flash demands of the weather station software do scale quite well with the amount of selected features. The "Barometer" configuration (P+Display), consisting of just an air pressure sensor and an LCD display, induces significantly smaller memory demands than the "Deluxe-PC" configuration (TWP+Serial+XML+Display) which bundles three sensors, a LCD display, and

[6] Tektronix TDS 2012, 100MHz resolution

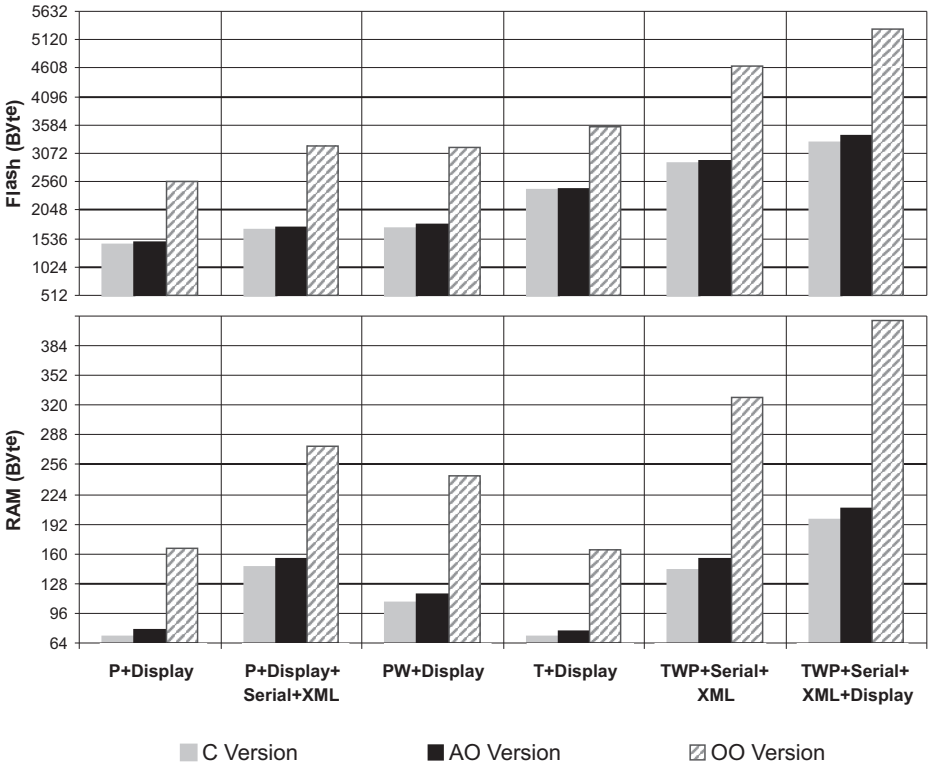

Fig. 7. Footprint comparison (RAM, flash) for different configurations

a XML-based PC connection over a serial interface. The memory requirements of the other examined configurations are in between. The noticeably high amount of flash memory required by the "Thermometer" configuration (T+Display) can be explained by the fact that this sensor is connected via the I^2C bus to the μ-controller (see Figure 2). To drive this bus, additional driver code is required that has not to be included for other sensors.

Overall, all three versions meet the goal of scalability, which is an indicator for achieved granularity. In every case, however, the OOP version requires significantly more memory space than its AOP counterpart which comes very close to the C version. Depending on the configuration, the required amount of RAM is up to 138% higher in the OO version, while the AO version takes only an extra of 10% at maximum (up to 13 byte)—both compared to the C-based version that does not provide SoC. The amount of flash memory is up to 91% higher in the OO version, but only 4% at maximum in the AO version. The net difference between using AOP and OOP has to be considered as even higher, as both versions of each configuration are linked with the same (configuration-

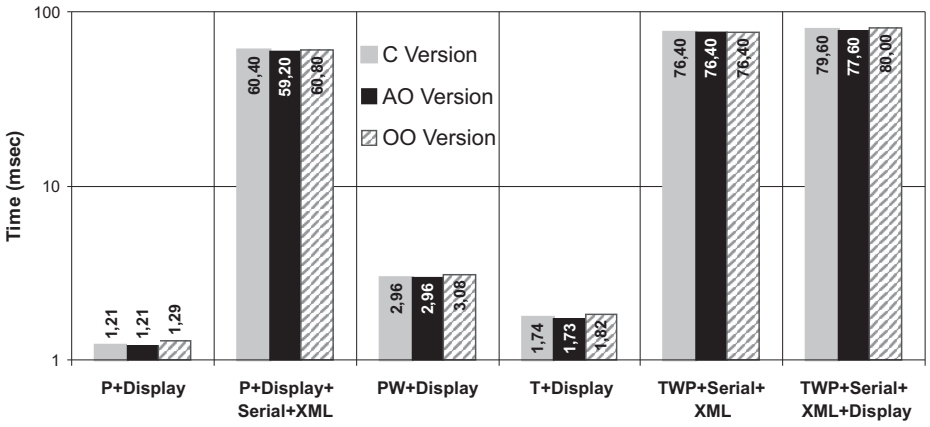

Fig. 8. Footprint comparison (time taken by a `Weather::measure()` / `Sink::process()` cycle) for different configurations.

dependent) set of device drivers whose memory requirements are included in these numbers.[7]

The runtime of all three versions is almost the same in all configurations (see Figure 8). It is mostly dominated by hardware and driver-related costs.

4.3 Memory Requirements in Detail

Table 1 breaks down the required overall amount of RAM and flash memory into their origins. It is evident, that the OO variants induce especially higher static memory demands. The *text* sections of the OO variants are up to 78%, the *data* sections up to 284% bigger than in the AO variants. The following cost drivers can be mainly accounted for this:

virtual functions are the main source of additional code and data, as they induce overhead on both, the caller and the callee side. On the caller side, each call to a virtual function requires, compared to advice code inlining, at least 16 additional byte in the machine code. On the callee side, virtual function tables have to be provided (4 byte + 2 byte per entry, *data*), object instances need to carry a pointer to the virtual function table (2 byte per instance), and constructors to initialize the vtable pointer of an instance have to be generated (at least 24 byte per class, *text*). In the "Barometer" configuration (P+Display), for instance, 52 byte of the *data* section are occupied solely by virtual function tables.[8]

Regarding code size, the situation may become even worse in larger projects: Due

[7] As parts of the driver code are implemented as inline functions, it is not possible to differentiate between driver-induced and application-induced code here.

[8] The AVR RISC core uses a Harvard architecture, thus vtables can not be placed in the text section.

Table 1. Memory usage and runtime of the AO and OO versions for different configurations

Configuration		text	data	bss	stack	flash	ram	time
P+Display	**C**	1392	30	7	34	1422	71	1.21
	AO	1430	30	10	38	1460	78	1.21
	OO	2460	100	22	44	2560	166	1.29
P+Display+	**C**	1578	104	7	34	1682	145	60.40
Serial+XML	**AO**	1622	104	12	38	1726	154	59.20
	OO	3008	206	26	44	3214	276	60.80
PW+Display	**C**	1686	38	14	55	1724	107	2.96
	AO	1748	38	18	61	1786	117	2.96
	OO	3020	146	33	65	3166	244	3.08
T+Display	**C**	2378	28	8	34	2406	70	1.74
	AO	2416	28	11	38	2444	77	1.73
	OO	3464	98	23	44	3562	165	1.82
TWP+Serial+	**C**	2804	90	17	35	2894	142	76.40
XML	**AO**	2858	90	23	41	2948	154	76.40
	OO	4388	248	39	41	4636	328	76.40
TWP+Serial+	**C**	3148	122	17	57	3270	196	79.60
XML+Display	**AO**	3262	122	24	63	3384	209	77.60
	OO	5008	300	44	67	5308	411	80.00

flash := text + data ram := bss + stack + data (Bytes) (ms)

to late binding, the bodies of virtual functions are never removed from the final image by means of function-level linking. Thus, “dead” function code that is never called at runtime becomes nevertheless part of the *text* section.

dynamic data structures are another source of additional overhead. The chaining of actors and sensors induces 8 additional data byte in the “Barometer” configuration, plus some extra code to access and iterate over the lists.

global instance construction causes some more “hidden” code to be generated. For each translation unit that defines one or more global objects, the compiler has to generate a specific initialization-and-destruction function (88 bytes, *text*). Pointers to these functions are additionally stored in the *data* section.

Regarding dynamic memory usage (*stack*), the differences between the AO and OO version are less significant. The OO variants need a few byte (up to 16%) more stack space than the related AO variants. This seems surprising at first, given that virtual function calls can not be inlined, therefore lead to a higher call depth and, thus, higher stack utilization. Part of this effect can be explained by the fact that the AOP version requires some additional stack space as well, namely by context-binding pointcut functions and the join point API (2-4 byte). The main reason is, however, that the maximum virtual function call depth is with 2 levels quite low. As the AVR architecture provides 32 general-purpose registers, which are also used for passing function parameters, it can furthermore be considered as quite “stack-friendly”. On other CPU architectures (such as Intel), the differences between AO and OO based solutions would be more significant.

4.4 Runtime Requirements in Detail

Table 1 also lists the measured runtimes in detail. In all configurations that support a serial connection the measurement/processing cycle time is mainly dominated by the underlying serial device driver. Here the costs are almost the same. In all other configurations the performance of the AO version and our highly efficient C implementation are the same. The runtime overhead of the OO version is between 4% and 6.6%. It can be explained with the numerous virtual function calls on the application level.

4.5 Cost Analysis Summary

Both, the AO and OO version of our product line, scale quite well. The runtime differences are small and only of minor importance in this domain. The OO version, however, induces dramatically higher memory requirements. Given a hardware product line like the AVR ATmega series, these differences can directly be mapped to the required μ-controller features. As Figure 9 shows, the hardware costs of the C-based and AO-based product line would be exactly the same in all configurations. Using AO versus OO leads even for our small example product line to significant differences regarding hardware costs. This effect would probably be even higher for larger product lines.

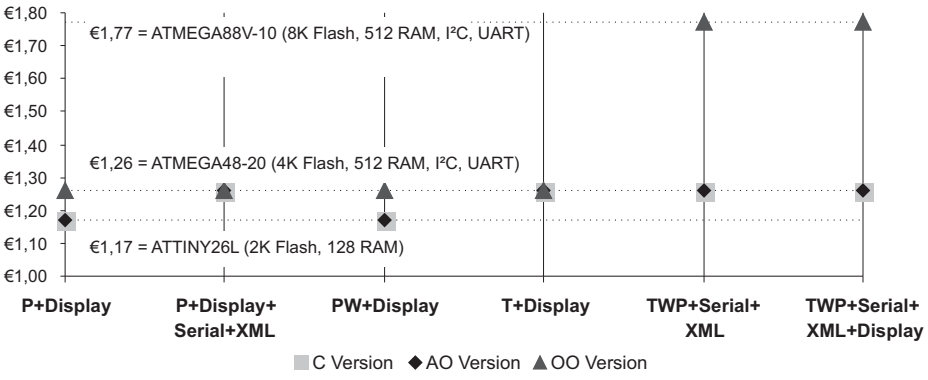

Fig. 9. Hardware scalability (required μ-controller variant) of the AO and OO versions for different configurations (wholesale prices: Digi-Key Corporation, Product Catalogue, Summer 2005)

5 Discussion

As pointed out in the previous sections, the AOP-based version of the weather station product line combines good SoC with high memory efficiency. In the following we discuss the principal reasons for this effect, the potential disadvantages, and to what degree our results are applicable to (system software) product lines in general and achievable with other AOP approaches.

5.1 General Advantages of the Approach

To ensure variability and granularity in product lines, components need to be implemented in a fine-grained and loosely coupled way. In our case study, we identified three fundamental concepts that are required to reach this: *Abstraction from concrete components* (interfaces), *abstraction from component sets* (1:n relationships), and *abstraction from component integration* (self-pluggability). The major disadvantage of OOP is that the features it provides to reach these goals are basically runtime concepts, while with AOP (in the case of static weaving) it is possible to use compile-time concepts:

abstraction from concrete components requires to use interfaces and runtime binding in OOP (thus, virtual functions in C++), while in AOP advice code is woven at compile-time to the join points (and in AspectC++ is even inlined).

abstraction from component sets requires to use some kind of runtime data structures in OOP, such as linked lists, while in AOP multiple advice can be woven at the same join point. Moreover, by means of aspect ordering, it is even possible to define a context-dependent invocation order.

abstraction from component integration is generally difficult to achieve in most OOP languages. In C++ it requires some constructor trickery to perform the integration at runtime. In AOP, Advice is *per se* an integration mechanism and resolved at compile-time. Moreover, aspects provide means for context-sensitive integration: Giving advice for a specific join point can be understood as a *weak reference* to another component, as the advice is silently ignored if the join point does not exist.

5.2 Compiler Issues

As pointed out in the analysis, much of the extra overhead of the OO version is directly or indirectly induced by the costs of late binding, even though the extra flexibility at runtime is not required in every case. A lot of work has been conducted to reduce the costs of C++ virtual functions[3, 8] and late binding in general, e.g. by static whole program optimization [18]. This gives rise to the question, if and to what extend the measured OO overhead could be prevented by better compilers.

Especially in the context of the highly complex and non-academic programming languages C and C++, which are the state of the art in system software and embedded software development, the implementation of optimizations based on static whole program analysis is extremely difficult. Although many of the aforementioned algorithms and techniques were proposed at least 10 years ago, today's commercial compilers have not incorporated these innovations yet. The main reasons are their complexity and existing de-facto standards like object file formats and ABIs.[9] Therefore, we pragmatically advocate for using OO and AO mechanisms in combination in order to get the best out of both worlds. While OO is well suited for applications that actually need dynamism, static advice weaving is an interesting alternative that avoids the OO overhead from the beginning if dynamism is not required.

[9] ABI = *Application Binary Interface*

5.3 Applicability to System Software

Compared to application software product lines, system software product lines have to fulfill even higher demands on configurability and resource-efficiency. With respect to efficiency, the above mentioned abstractions are particularly useful in system software:

efficient callbacks by advice code inlining. System software usually has to provide means to bind user-defined code to (asynchronous) system events, such as hardware interrupts in an operating system. To prevent uses-dependencies[41] of the system code to driver/application code, a *callback* or *upcall* mechanism via function-pointers has to be provided. This leads to extra overhead and higher latencies at runtime. As aspects reverse such dependencies[14], they provide means to break uses-dependencies without this extra overhead.

handler chains by multiple advice. The implementation of handler chains can be implemented by the same mechanism as single callbacks. Due to advice code inlining, this induces furthermore no additional overhead.

configurable initialization order by aspect precedences. In system software it is often necessary to initialize components and subsystems in some well-defined (partial) order during system startup. The order is determined by component-interdependencies, however, not necessarily fixed in the case of configurable product lines. It may well depend on the actual configuration. By aspect precedences (given as *order advice* in AspectC++), aspects provide perfect means to solve this problem at compile-time.

5.4 Potential Disadvantages of the Approach

The advantages regarding resource efficiency are mainly caused by binding and inlining component code at compile-time that is bound at runtime in OOP. This may, however, also lead to limitations of the approach:

compile-time fixation limits the approach to static configuration of product lines. In domains, where the set of selected features can arbitrary change at runtime, additional support has to be provided e.g. by a dynamic aspect weaver. As most dynamic weaving approaches induce inevitable overhead[27], it is questionable if in such cases AOP still out-weights the overhead of OOP. However, in most cases only some features can be expected to change at runtime. A tailorable low-cost weaving infrastructure, which provides means to configure *per-feature* if static or dynamic weaving should be used [45], may be a promising solution for such cases, as this enables developers to optimize the trade-off between dynamism and resource overhead at configuration time.

code bloating due to advice code inlining. This occurs if functionality given by advice is invoked from (and therefore woven to) many different join points. While this situation has to be considered for highly cross-cutting concerns such as tracing, component composition usually takes place at few well-known join points only. In any case, however, the programmer is able to prevent this effect by simply moving the advice body into a non-inlined function that is invoked from the advice code.

5.5 Applicability to Other AOP Approaches

Our results show, that in the case of C++ and AspectC++, using AOP instead of OOP may lead to significant saving effects in terms of memory and hardware costs. This gives rise to the question, if similar results can be reached with other languages and/or AOP approaches as well. This is clearly a huge field for further research and can not be answered in this paper. We think, however, that the following points are of particular importance:

weaver runtime support may lead to some unavoidable base overhead that limits the reachable amount of memory savings. This is clearly an issue for the domain of embedded systems. Experiments conducted with AspectJ on Java2 Micro Edition, for instance, have shown that the AspectJ runtime induces an extra 35KB overhead[49]. AOP approaches that do not require runtime library support, are probably better suited for the goal of resource-thriftiness. Besides AspectC++, XWeaver[44] and AspectC[12], for instance, may be promising candidates.

language capabilities of the host language have a high influence on the applicability of the approach. As the resource savings are mainly caused by using compile-time instead of runtime concepts, the host language has to support such concepts. On the one hand, it is probably difficult to get "rid of the OOP overhead" in languages that have been particularly designed as object-oriented languages, such as Java or C#. On the other hand, languages such as C may not be sufficiently expressive with respect to (compile-time) genericity to take full advantage of the approach.

5.6 Design Issues

The actual design and implementation of the AO and OO versions have a significant influence on the resulting memory and performance numbers. Both versions have achieved the *granularity*, *pluggability*, and *extensibility* goals. One might, however, question if especially the design of the OO version is optimal with respect to the *economy* goal as well. Theoretically, it might be possible to find a better OO implementation that leads to lower memory demands.

As a matter of fact, it is impossible to proof that the design and implementation of some software is optimal with respect to runtime resource requirements. The authors have, however, profound experience in developing resource-minimal object-oriented product lines[11, 10, 34]. There is, furthermore, quite some evidence that any OO version would lead to higher memory requirements than the AO version: As pointed out in the previous sections, runtime mechanisms have to be used with OO to reach the required flexibility which, as described in sections 3.5 and 4, induce overhead. Given that the OO version takes up to 138% more RAM and up to 91% more flash memory than the C and AO versions, it is hard to imagine that these differences could be levelled by a better OO implementation.

For the AO version the resource requirements are only minimal higher than for the C version. Hence, it can already be considered as "nearly optimal". In a recent study on the eCos operating system product line[1], we could show that the C-mechanism to reach configurability (conditional compilation), can be replaced by AOP concepts without any

extra runtime and memory costs. In this study, we refactored several hundred `#ifdef`-blocks caused by configurable features into aspects[33].

6 Related Work

Several papers have been published that analyze and compare different implementation techniques of product line variabilities[23, 36]. All authors come to the conclusion that AOP is a very promising approach, although still not widely used in the industrial practice[38]. In the middleware area several case studies showed that AOP allows software developers to scale features and footprint by supporting the static configuration of fine-grained configurable features[50, 13, 51]. All of these studies were conducted with AspectJ. In one case the authors clearly expressed that AspectC++ would be more appropriate for the embedded systems domain[28].

Other proposed implementation techniques were Generative Programming[16], the application of frame processors such as XVCL[29] or Angie[19], and Feature-Oriented Programming (FOP)[6]. All three techniques have in common with AOP that in a layered system design the higher-level layers can use special means to refine the behavior and/or structure of the lower-level layers. This *dependency alignment*[14] is a key to successful product line design. In comparison to frame processing and C++ template-based generative programming[17], the main advantage of the AOP approach is its *obliviousness*, i. e. refined layers don't have to be prepared for intervention by aspects. In comparison with refinements in FOP, the advantage of AOP is *quantification*, i. e. a single aspect can affect various different join points. Of course, FOP is much more than just the refinement mechanism. Therefore, the novel FeatureC++ language[5] is very promising, as it combines AOP and FOP.

When using OOP, the discussed refinement can only be achieved with virtual functions at the cost of dynamic binding. However, in many cases dynamic binding offers more flexibility than it is actually needed. A lot of work has been conducted to reduce the costs of virtual functions[3, 20, 8], but these optimizations are still not state of the art in current C++ compilers.

Another alternative approach is to combine OOP with partial program evaluation tools like Tempo [15] and C-Mix [4], especially as partial evaluation has succesfully been applied in the context of system software [37]. However, the authors are not aware of any tools or concept papers that cover the combination of these techniques with OOP. Furthermore, the case studies on partial evaluation of system software focus on performance improvements. Some of these techniques increase the code size by keeping specialized and original code in memory at the same time or by applying dynamic compilation [42].

In the systems software area configurable software is often component-based. Popular examples are the OSKit[22, 39] and TinyOS[7]. In both cases the requirements on performance and footprint can only be achieved with additional tool support. For example, Knit can be regarded as a weaver for OSKit components[43]. TinyOS is written in the NesC language[25]. Here the compiler performs a whole program transformation in order to optimize the system. In both cases the overhead, which is the result of the

component model, is later removed by an additional tool. With the AOP approach an overhead is avoided from the beginning.

An interesting alternative to layered designs is Subject-Oriented Programming (SOP)[26, 40]. This technique is one of the roots of AOP. It supports system composition from *slices* instead of layers. In our embedded weather station case study, the sensors are in fact slices. By using the introduction mechanism of AspectC++, SOP was simulated.

7 Summary And Conclusions

The development of fine-grained and resource-efficient system software product lines requires means for separation of concerns that do not lead to an extra overhead in terms of memory and performance. Especially in the important domain of small embedded systems, where even a few additional bytes of code or data may lead to higher hardware costs, the inherent overhead of OOP is not acceptable for most developers. For this domain we propose AOP as an alternative.

In this paper, we compared an object-oriented with an aspect-oriented implementation of an configurable embedded software product line. We could show that the aspect-oriented implementation induces significantly lower memory requirements than the object-oriented implementation, while providing similar or even better qualities with respect to separation of concerns. Although our results were produced with C++ and AspectC++, similar effects should be achievable with any AOP approach that does not require additional runtime support and performs inlining of advice code. Advice code inlining also leads to an excellent performance. Our measurements have shown that the performance of the AO version was identical with the performance of our C-based reference implementation.

Overall, our results show that aspects provide very good separation of concerns in conjuction with high resource efficiency. This makes AOP well suited for the development of embedded system software product lines – where aspects beat objects.

Regarding future work, we will continue to evaluate costs and benefits of applying AOP to the operating systems domain. We are furthermore working on a benchmark suite that provides detailed results regarding the runtime and memory costs of AspectC++ features.

8 Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. We furthermore thank Danilo Beuche from pure::systems GmbH for leaving us the weather station hardware. Finally, a big thankyou goes to the editors, especially Hans-Arno Jacobsen, whose suggestions helped us a lot to improve the quality and comprehensibility of this paper.

This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4 and SP 968/2-1.

References

1. eCos homepage. http://ecos.sourceware.org/.
2. OSEK/VDX standard. http://www.osek-vdx.org/.
3. Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. Technical Report TRCS95-22, Computer Science Department, University of California, Santa Barbara, December 1995.
4. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
5. Sven Apel, Thomas Leich, Marko Rosenmller, and Gunter Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE '05)*, Tallinn, Estonia, September 2005.
6. Don Batory. Feature-oriented programming and the AHEAD tool suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 702–703. IEEE Computer Society, 2004.
7. UC Berkeley. TinyOS homepage. http://www.tinyos.net/.
8. David Bernstein, Yaroslav Fedorov, Sara Porat, Joseph Rodrigue, and Eran Yahav. Compiler optimization of C++ virtual function calls. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS '96)*, Toronto, Canada, June 1996.
9. Danilo Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.
10. Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.
11. Danilo Beuche, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Streamlining object-oriented software for deeply embedded applications. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS '00)*, pages 33–44, Mont Saint-Michel, France, June 2000.
12. Yvonne Coady, Gregor Kiczales, Michael Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 3rd Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE '01)*, 2001.
13. Adrian Colyer, Andy Clement, Ron Bodkin, and Jim Hugunin. Using AspectJ for component integration in middleware. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 339–344, New York, NY, USA, 2003. ACM Press.
14. Adrian Colyer, Awais Rashid, and Gordon Blair. On the separation of concerns in program families. Technical Report COMP-001-2004, Lancaster University, 2004.
15. C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E.-N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es), 1998.
16. Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.
17. Krzysztof Czarnecki and Ulrich W. Eisenecker. Synthesizing Objects. In R. Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, number 1628 in Lecture Notes in Computer Science, pages 18–42, Lisbon, Portugal, 1999. Springer-Verlag.

18. Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of PLDI '95*, La Jolla, CA, June 1995.
19. Delta Software Technology GmbH. Angie – an introduction, June 2005.
20. Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*, October 1996.
21. Bruno Dufour, Christopher Goard, Laurie Hendren, Clark Verbrugge, Oege de Moor, and Ganesh Sittampalam. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 150–169, New York, NY, USA, 2004. ACM Press.
22. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for Kernel and language research. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '97)*, ACM Operating Systems Review, pages 38–51. ACM Press, October 1997.
23. Cristina Gacek and Michalis Anastasopoules. Implementing product line variabilities. In *Proceedings of 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, pages 109–117. ACM Press, 2001.
24. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
25. David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, pages 1–11, San Diego, CA, USA, 2003. ACM Press.
26. William Harrison and Harold Ossher. Subject-oriented programming—a critique of pure objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, September 1993.
27. Michael Haupt and Mira Mezini. Micro-measurements for dynamic aspect-oriented systems. In *NetObjectDays (NODe '04)*, volume 3263 of *Lecture Notes in Computer Science*, pages 81–96, Erfurt, Germany, September 2004. Springer-Verlag.
28. Frank Hunleth and Ron Cytron. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the 2002 Joint Conference on Languages, Compilers and Tools for Embedded Systems & Soft. and Compilers for Embedded Systems (LCTES/SCOPES '02)*, pages 38–45, Berlin, Germany, June 2002. ACM Press.
29. Stan Jarzabek and Hongyu Zhang. XML-based method and tool for handling variant requirements in domain model. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE '01)*, pages 116–123, Toronto, Canada, August 2001. IEEE Computer Society Press.
30. J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *International Conference on Mobile Computing and Networking (MOBICOM '99)*, pages 271–278, 1999.
31. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, June 2001.
32. Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 55–74. Springer-Verlag, October 2004.

33. Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the EuroSys 2006 Conference (EuroSys '06)*, pages 191–204. ACM Press, April 2006.
34. Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. On the design and development of a customizable embedded operating system. In *Proceedings of the SRDS Workshop on Dependable Embedded Systems (SRDS-DES '04)*, pages 1–6. IEEE Computer Society, October 2004.
35. Daniel Lohmann and Olaf Spinczyk. On typesafe aspect implementations in C++. In F. Geschwind, U. Assmann, and O. Nierstrasz, editors, *Proceedings of Software Composition 2005 (SC '05)*, volume 3628 of *Lecture Notes in Computer Science*, pages 135–149, Edinburgh, UK, April 2005. Springer-Verlag.
36. Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings of ACM SIGSOFT '04 / FSE-12*, November 2004.
37. G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the sun commercial rpc protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–125, Amsterdam, The Netherlands, June 1997. ACM Press.
38. Dirk Muthig and Thomas Patzke. Generic implementation of product line components. In *NetObjectDays (NODe '02)*, volume 2591 of *Lecture Notes in Computer Science*, pages 313–329, Erfurt, Germany, October 2003. Springer-Verlag.
39. University of Utah. OSKit homepage. http://www.cs.utah.edu/flux/oskit.
40. Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, pages 43–50, October 2001.
41. D. L. Parnas. Some hypothesis about the uses hierarchy for operating systems. Technical report, TH Darmstadt, Fachbereich Informatik, 1976.
42. Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.
43. Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 347–360, San Diego, CA, USA, October 2000. Usenix Association.
44. O. Rohlik, A. Pasetti, V. Cechticky, and I. Birrer. Implementing adaptability in embedded software through aspect oriented programming. In *Proceedings of Mechatronics & Robotics (MechRob '04)*, Aachen, Germany, September 2004. IEEE Computer Society Press.
45. Wolfgang Schröder-Preikschat, Daniel Lohmann, Wasif Gilani, Fabian Scheler, and Olaf Spinczyk. Static and dynamic weaving in system software with AspectC++. In Yvonne Coady, Jeff Gray, and Raymond Klefstad, editors, *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS '06) - Mini-Track on Adaptive and Evolvable Software Systems*. IEEE Computer Society Press, 2006.
46. Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in AOP with AspectC++. In Hamido Fujita and Mohamed Mejri, editors, *New Trends in Software Methodologies, Tools and Techniques (SoMeT '05)*, number 129 in Frontiers in Artificial Intelligence and Applications, pages 33–53, Tokyo, Japan, September 2005. IOS Press.
47. David Tennenhouse. Proactive computing. *Communications of the ACM*, pages 43–45, May 2000.
48. Mark Weiser. The computer for the 21st centrury. *Scientific American*, 265(3):94–104, 1991.
49. Trevor Young and Gail Murphy. Using AspectJ to build a product line for mobile devices. Chicago, Illinois, March 2005. AOSD '05 Demo.
50. Charles Zhang, Depend Gao, and Hans-Arno Jacobsen. Generic middleware substrate through modelware. In *Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference (Middleware '05)*, Grenoble, France, 2005. ACM Press.

51. Charles Zhang and Hans-Arno Jacobsen. Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11), November 2003.