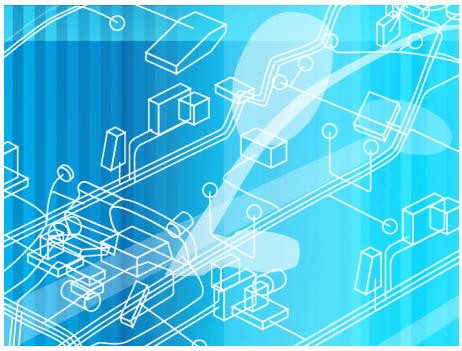
Fraunhofer Institut Software- und Systemtechnik

Modeling Overlapping Functionality

Ekkart Kleinod



ISST-Bericht 88/08
August 2008

Editor: Fraunhofer-Gesellschaft e. V.
Institute for Software and Systems Engineering
Director: Prof. Dr. Jakob Rehof
Berlin branch of the institute: Mollstrasse 1
10178 Berlin
Germany
Dortmund branch of the institute: Joseph-von-Fraunhofer-Strasse 20
44227 Dortmund
Germany

ISSN 0943-1624

The Research Project VEIA

The project VEIA (Distributed Development and Integration of Automotive Product Lines) is supported by the German Federal Ministry of Education and Research within the "Forschungsoffensive Software Engineering 2006", Project No. "01ISF15A".

The following four partners are engaged in the project:

- BMW Group
- Fraunhofer ISST
- PROSTEP IMP GmbH
- Technische Universität München

Website: http://veia.isst.fraunhofer.de/

Author

Ekkart Kleinod (Fraunhofer ISST).

Abstract

The reference process for the development of automotive product lines developed in the project VEIA introduces different kinds of models for the representation of architectural views of the system family under development. So far we described the different models for these views and guidelines on how to develop the models.

In this paper we describe a solution for the problem of representing overlapping functionality in component architectures. The approach uses aspect-oriented concepts to describe overlapping functionality by components without overlaps. The resulting system architecture is generated by a weaving algorithm.

The approach moreover contains concepts for further abstractions of aspects and their instances that support a patternlike solution for the problem of reusing functionality.

Contents

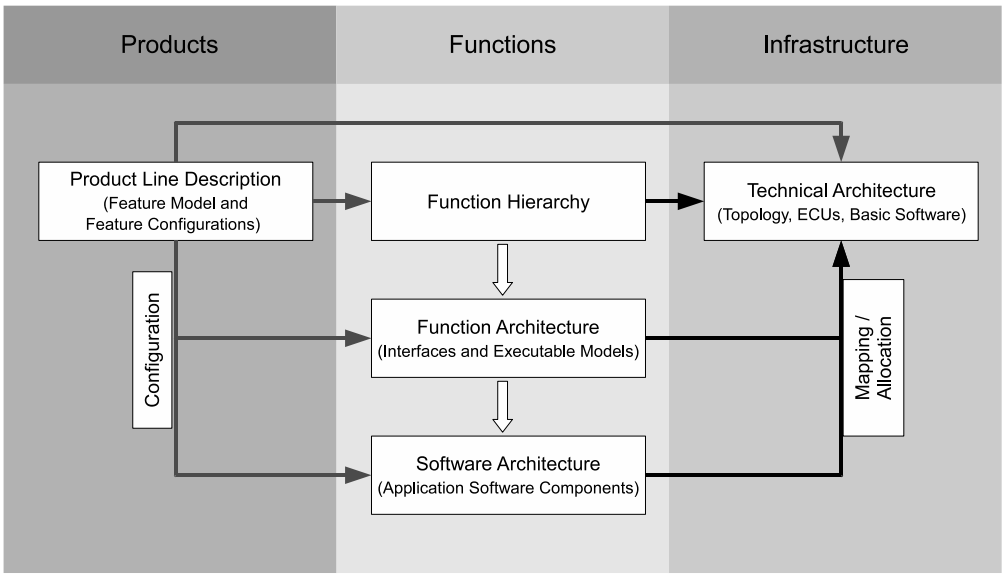
1	**Introduction**	5
1.1	Area of modeling	5
1.2	Problem	6
1.3	Approach	8
1.4	This document	10
2	**Problem**	11
2.1	Case study "Condition Based Service"	11
2.2	Modeling CBS in VEIA	11
2.3	Exemplary function architecture	13
2.4	Scoping	18
2.5	Systematization of the problems	20
3	**Related solutions and work**	23
3.1	Component-oriented cuts	23
3.2	Secondary hierarchies	27
3.3	Aspect-orientation	28
4	**VEIA approach**	35
4.1	Aspect-oriented modeling	35
4.2	Aspect-oriented modeling (case study CBS)	36
4.3	Aspect and instance models	40
4.4	Patterns	47
4.5	Formalization	51
5	**Discussion**	55
5.1	Overall discussion	55
5.2	Open issues	56
5.3	Related work	57
5.4	Pro and Contra	59
References		61

1 Introduction

1.1 Area of modeling

In project VEIA a reference process for the development of automotive product lines has been devised. It describes three basic views that have to be modeled within the development of an automotive product line: product models, function models, infrastructure models (see Figure 1 and [Gro08]).

Figure 1 VEIA reference process survey.



Every view contains one or more artifacts that model the automotive system. The functional view, for instance, contains three main artifacts: the function hierarchy, the function architecture, and the software architecture. All models are interconnected by relations, either refinement relations, configuration relations, or allocation relations.

The models, that make the views and therefore the system, are described using different modeling techniques. The product line description, for example, uses feature trees as modeling language. The other models of the function and infrastructure view use hierarchic component architectures as means of modeling.

Hierarchic component architectures consist of components, that can be decomposed in smaller components or component architectures, respectively. Every component consists of an interface, that defines the boundaries of the component, the black box view. The internal structure, and behavior is defined in the body of a component.

Because of the similarity of component architectures by means of structuring and composition/decomposition concepts, in the following, only the function architecture is discussed. All problems and solutions are valid for the other component architectures as well, although it is possible, that there are small differences in the implementation of the solutions.

1.2 Problem

In automotive modeling, the target system is the electronic system of a whole car. Therefore the function architecture models the whole system. In order to get manageable and processable subsystems, the function architecture is decomposed in subsystems, that are further decomposed etc.

This decomposition takes place according to different criteria, e.g. functionality, commonality, user visibility, or organizational needs. Often the chosen decomposition criterion does not provide good means to uniquely separate two parts. A steering wheel, for example, can be chosen to belong to the driving features, it can represent part of an MMI[1], or it can be classified as part of the navigation system.

This means, depending on the decomposition criterion, different architectures are created during the decomposition process. Every architecture has its advantages and disadvantages concerning one or more decomposition criteria. One common criterion for the first decomposition steps is the organizational structure, in which the subsystems are implemented. If, for example, the steering wheel is modeled as part of the MMI, it can be directly identified as such part of the MMI, therefore the responsibility for its implementation is put on the MMI unit[2]. On the other hand, the unit responsible for the driving features has no direct access to the driving wheel, they have to communicate and cooperate with the MMI unit in order to

[1] MMI: man machine interface
[2] The term "unit" is used to denote a project team, be it an organizational unit or not.

model and implement their system. A more elaborate example for this problem is given in the next section using the case study "Condition Based Service".

The main problem is the need to chose between one criterion or the other. Component architectures do not allow components to hierarchically belong to several components at the same time, they are disjoint entities of an architecture. Therefore, when modeling intertwined functionality with component architectures, compromises have to be accepted. This leads to loss of modeling information, because the interdependencies are not modeled. Furthermore parts of components are scattered throughout other components, which leads to tangled models in other components. In conclusion, overlapping functionality leads to crosscutting components, that have to be dealt with. These crosscutting components are not well modeled concerning coherence and cohesion, therefore resulting in several interdependencies of the architecture.

However, it is not the goal of this paper to challenge the modeling principle of component architectures. On the contrary, the principle of disjoint components leads to hierarchically well structured models, that are easy to maintain. The advantages of component architectures are manifold: they reduce the modeling complexity, or provide stable, reusable components.

The problem of avoiding overlapping components manifests in different ways. In this paper we describe two problems in detail: hierarchic composition of components and scoping/cutting of components.

The first problem, hierarchic composition of components, deals with the methodical question, which subcomponents should be grouped hierarchically into one component. Seen from the other side of the model, the question is, how to decompose one component into subcomponents. In relation to the problem of overlapping components, this problem mainly occurs in the early stages of system design, when overall decisions are made. One example is the first hierarchy of organizational units, that has to be decomposed into the main car functions. Nearly every main car function can be grouped to several organizational units, but only one unit can be chosen.

The second problem, scoping/cutting of components, describes a rather "technical" problem. Regardless on how the components are decomposed, they still have to communicate with each other. Depending on the decomposition, the interfaces and connectors have to be modeled accordingly. Therefore the whole system communication depends on the decomposition of the system. In order to model an overlap-free system, the function scope has to be aligned with the component scope. This alignment bases on design decisions, that are not kept in the model themselves. Thus, these design decisions are lost after the scoping process is finished.

Overlapping concerns themselves are no fundamental problem for modeling systems. If there was a way to describe and handle architectures with overlapping concerns, there would be no problem at all.

On the other hand, the development of large, complex systems, and distributed development as well, are far more easy to maintain, when hierarchic modeling techniques are used. This was mainly introduced in programming and modeling by David Parnas in his paper on decomposition criteria [Par72]. Hierarchic decomposition for example lead to component-oriented modeling languages. Hierarchic decomposition, on the other hand, demands for disjoint hierarchic elements: e.g. components.

Thus, until the late 1990s, these overlaps in concerns could not be modeled, because of the disjoint decomposition techniques. Therefore, in the following discussion, we relate to component-oriented techniques to describe the problems of overlapping concerns.

When decomposing a system into hierarchic components, the problem of overlapping concerns has to be dealt with pragmatically. Component modeling does not provide the means of describing these concerns in their overlapping nature. Aspect-oriented techniques, or subject-oriented modeling methods try to solve the problem by allowing native modeling of overlapping concerns. In general, one can say, there is a conflict between concerns and components, when concerns have to be described as components.

Concluding, the problem of overlapping functionality leads to several problems on several levels of modeling. The main reason for all problems is based on the fact, that overlapping functionality cannot be modeled by means of component architectures with disjoint hierarchical components. Therefore compromises have to be made concerning the system design, that lead to the aforementioned problems.

1.3 Approach

In this document we introduce an approach to model overlapping components natively, i.e. to provide means for modeling overlapping components within component architectures. Additionally, we describe a weaving algorithm for converting overlapping architectures to regular component architectures with disjoint components. Thus, the new approach can be used for modeling a system with overlaps without losing connection to existing processes and models, that work with component architectures.

The approach uses aspect-oriented principles in order to describe overlapping functionality. Within the aspect-oriented community, problems such as crosscutting components, scattering, or tangling are the main focus of research. Originating in support of programming languages, aspect-orientation spread out to other areas of system design, such as requirements engineering, modeling, or code generation. The aspect-oriented solutions so far deal with models in a way, that supports code generation or modeling of programs in UML. There are but a few approaches for high-level modeling such as modeling in architecture description languages (ADLs). Among these approaches even less tackle special problems of the automotive industry, such as distributed development processes, integration of suppliers, preservation of intellectual property (IP), seamless integration in existing processes and model landscapes etc.

Therefore, a new approach, that bases on existing solutions, but focuses on the automotive domain is needed in order to introduce a practical approach into existing development processes. The challenge is not only to solve the aforementioned problems, but to define an approach that is applicable in practice, that respects existing models, experiences, processes etc.

The presented approach allows for these conditions by providing not only means for new modeling techniques but by providing a solution to generate well-known models, that can be used as before. The new modeling techniques are, first, the possibility to model overlapping functionality. Secondly, the approach regards the need for pattern solutions by introducing the concept of aspects and their instances. This results from the experiences with the approach so far. Often, functions not only overlap but consist of a central part and distributed parts, that are alike. The parts that are alike build patterns, that are used several times. Examples are "condition based service", "error logging", or introduction of safety patterns such as redundancy. Our approach separates these patterns from their use in the model. Thus, the complexity of the models can be reduced without reducing the modeling power. Furthermore, the patterns are modeled as overlapping functionality, therefore allowing the weaving algorithm to generate the component architecture, that can be used as before.

As mentioned before, the approach concentrates on function architectures as example for component architectures. The adoption of the approach to other architecture description languages, e.g. for the infrastructure is an open issue. The concepts, however, are easily adoptable, for their simplicity. For the moment, the approach is defined and formalized for the structure of the models. It does not solve the problem of behavioral description and weaving. Nevertheless, the problems and possible solutions for behavioral modeling are discussed throughout the document.

1.4 This document

This document describes the aforementioned approach in the context of the project VEIA. First, the problem is discussed in the context of the case study *Condition Based Service (CBS)*. Section 2 introduces the case study, afterwards the modeling shortcomings are described and displayed. A short systematization of the problems closes the section.

Secondly, existing solutions are discussed. In Section 3 we first discuss the traditional way of component-oriented cuts in order to create disjoint component architectures. After that, we dispute secondary hierarchies. The section is closed with the discussion of aspect-oriented approaches.

Thirdly, we present our approach. In Section 4 we introduce the aspect-oriented part of the approach. Afterwards, we extend the approach by the separation of aspects and their instances. We use this extension to describe the modeling of patterns and their introduction into the models. The sections ends with the formalization of the concepts.

Finally, the results and the approach are discussed in Section 5. After an overall discussion we name the open issues, that have to be solved later on. The dispute of pro and contra ends this section.

The end of the document is the references section.

2 Problem

This section describes the problem in more detail. We therefore introduce the case study "Condition Based Service". It will help to demonstrate the problem using a real life example.

2.1 Case study "Condition Based Service"

The case study "Condition Based Service" consists of a real world function, that is modeled throughout VEIA. The function "Condition Based Service" (CBS) itself is described and modeled in [GKM07b]. Therefore, we give only a short summary introduction into the function:

CBS is an auxiliary function in a car, that computes and displays the next service date. The computation bases on the wear-out of certain wearing parts, that is either measured with a sensor, or estimated based on average, empirical wear-outs of these parts. The parts that are included in the computation vary over the different products or product families. Premium cars, for example, contain more sensors, more parts are monitored, therefore, the computation is more accurate. Middle class cars monitor less parts, they too contain less sensors, therefore, the computation is less accurate, but still better than the non-adaptive computation so far.

2.2 Modeling CBS in VEIA

In VEIA, the reference process defines the artifacts, that have to be modeled for a system. In the case study, we modeled the product line description, the function architecture, the software architecture, and the technical architecture. Furthermore, we created the connecting models for configuration and allocation. Thus, the function CBS is modeled completely within VEIA ([GKM07b]).

During the modeling phases, we researched several methodical and conceptual issues. The first issue, of course, was the development of the VEIA reference process ([GEKM07]). We described three main views and the corresponding artifacts, that describe a system at a very abstract level. Variance handling and

modeling was integrated from the beginning, therefore allowing product families to be modeled with VEIA artifacts. In order to completely support product families, all artifacts contain means of describing, identifying, and configuring variance within the model.

Secondly, we researched measuring our models. Therefore, we researched metrics for architectures in general ([GKM07a]). During this work we had to acknowledge, that very few metrics exist, that measure not only architectures but architectures for product families. Therefore a main focus had to be laid on adopting and interpreting metrics in order to achieve meaningful results. This work was deepened in [Man08a], where the number of discussed metrics was increased and the conceptual foundations were widened. Both papers use CBS in order to measure an existing architecture and to answer questions concerning implementation costs, complexity, benefit of reuse etc.

The question of variant handling concerning configuration was thirdly researched in [MR08]. In this paper, configuration is discussed with respect to the impacts on all VEIA artifacts. Furthermore, questions like completeness of configuration, conflicts, conflict resolution etc. were raised and answered. The CBS function contains several variants, thus allowing the methods and concepts to be checked against a real system.

Last but not least, the process of modeling was researched in [Gro08]. In this paper, the modeling process of the case study is discussed and annotated. The main idea is modeling fast, in order to achieve quick results, but on the other hand to model as formal as possible. The paper discusses methods to achieve a good balance between these concurring goals. Again, CBS is the example, that is modeled throughout the paper, in order to consolidate the methodical steps.

One part of the modeling process is the definition of the scope of functions, and therefore, components. The issue of scoping arises throughout all modeling activities that concern component architectures. It was not researched systematically before in VEIA, scoping was done pragmatically, geared to existing solutions. The problem of scoping is part of the overall problem of overlapping functionality, that is discussed in this paper. In the following, the problem is shown in more detail than in the introduction. We use the example of "Condition Based Service" for the illustration of the problem. While this example is a real-life example, the solutions in this paper do not reflect real-life models, e.g. used by the BMW Group.

2.3 Exemplary function architecture

In the following we will model the function architecture of a car system. One of the functionalities will be "Condition Based Service". CBS is used as an example for an overlapping functionality, so we can illustrate the problem of component architectures with respect to overlaps.

One can model a system top-down or bottom-up, depending on the level of knowledge about the system, or organizational standards, or modeling guidelines, etc. In the automotive industry it is hard to motivate a top-down approach, because the systems already exist, so there is no need to start over new. On the other hand, that methodical drawbacks, such as the inability to model overlapping functionalities, are solved pragmatically in existing architectures, so there exists the need to revise previous design decisions. Therefore, we start to model the system not completely from the scratch, but not with an existing solution either. We first list the functionalities, that should be carried out in the system. After that we model these functionalities in a function architecture.

Before delving into modeling itself, we have to focus on decomposition, hierarchies and scoping. These problems are facets of the same modeling process: defining subsystems of a system or subsystem, respectively, so the modeling and implementation process can be continued.

The topmost component of a component architecture describes the system to be modeled. In order to model this component, the designer has to decide, what lies within the system and what lies outside, i.e. the designer has to define the scope of the system. In order to partition the system into subsystems, the designer has to define the scope for every subsystem as well. This continues for every hierarchic level that is modeled. In component architectures only disjoint components are allowed. This means, the scope for every component has to be disjoint as well.

The process of scoping is related to the definition of hierarchies. A hierarchy in a component architectures defines, which components are subcomponents of a hierarchic component. Thus, the hierarchic component consists of its subcomponents, adding no further behavior than defined in the subcomponents.[3] The definition of the hierarchy on a higher level often follows organizational needs, implementation responsibility etc. The hierarchies on lower levels are more function driven, as the

[3] This is a constraint, that is set for VEIA models. It is not necessarily true for other component architecture models.

components tend to become implementable elements.

The definition of hierarchies in a top-down-approach is called decomposition. The process in a bottom-up-approach is called composition. Decomposition and composition are terms for the process of defining hierarchies. Often the term "decomposition" is used for the result of the decomposition process: a structure decomposed into components. In this document we will separate the models (subcomponents) from the process (decomposition) for the sake of clarity.

A decomposition has to follow certain decomposition rules, that have to be set before or during the process. Possible criteria for decomposition are: functionality, commonality, user visibility, organizational needs, or even rules of thumb. As mentioned before, the criterion can depend on the level of hierarchy, that is achieved so far. Which criterion is chosen depends upon the needs and organization of the development process.

The main problem concerning the decomposition lies in the impossibility to create an overlap-free decomposition. Every decomposition criterion focuses on one main aspect of a system. Given a sufficiently large system, not every function will fit into one criterion. Several functions interact with each other, often functions are intertwined by their very nature. Thus, one criterion separates two functions, whereas the same criterion cannot separate a third function from the others. This problem is well known in modeling, even in 1972 David Parnas raised the question of good decomposition criteria [Par72]. Tackling the problem he introduced the concept of information hiding, i.e. creation of disjoint components. Now, we know, that information hiding helps organizing components and therefore helps modeling a system, yet the problem of overcoming the "Tyranny of the Dominant Decomposition" [IBM] is not solved. Although this cannot be scientifically proven for all decomposition criteria, there is enough experience in the modeling community, especially the aspect-oriented community, founding this thesis.

In the following we will model a car system with special focus to the example "Condition Based Service", in order to identify and illustrate the problems.

System functionality

The following list contains an extract of the functionality of a car system. We chose to model but a few functionalities for the sake of brevity. The system to be modeled consists of the following functionalities:

- ignition control
- throttle control
- motor oil check

- gear oil check
- driver's display
- time display
- kilometer reading display
- tachometer
- radio display
- alert warning display
- head-up display
- condition based service (inspections, wheels, motor oil, spark plugs, particle filter, brake pads, microfilter, brake fluid, gear oil)
- wheel pressure control
- wheel speed control
- brake pad check

First hierarchy

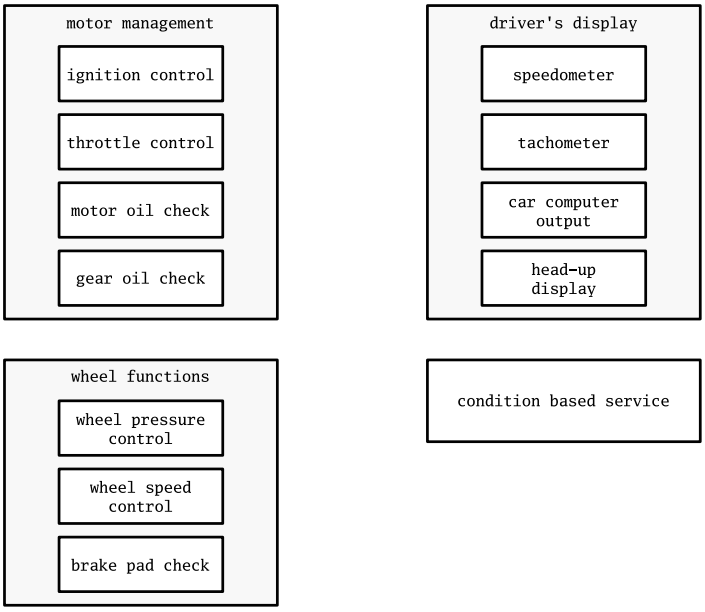
The first hierarchy reflects organizational concerns.[4] This is the designer's decision, it bases on our experiences with OEMs. The first hierarchy is the responsibility hierarchy, too. This means, the subsystems created here are given to the according organizational unit, which is responsible for the further modeling and implementation of the subsystem. The following hierarchy is created:

- motor management
  - ignition control
  - throttle control
  - motor oil check
  - gear oil check
- driver's display
  - speedometer
    - time display
    - kilometer reading
  - tachometer
  - car computer display

[4] Organizational units mostly reflect responsibilities that may arise from hardware views on the system. Other possible structures consider a process view, resulting in requirements units, design units, etc. In our example we assume a mixed hardware and functional view.

    - radio
    - alert warnings
  - head-up display
- condition based service (inspections, wheels, motor oil, spark plugs, particle filter, brake pads, microfilter, brake fluid, gear oil)
- wheel functions
  - wheel pressure control
  - wheel speed control
  - brake pad check

Figure 2 First hierarchy of the system.



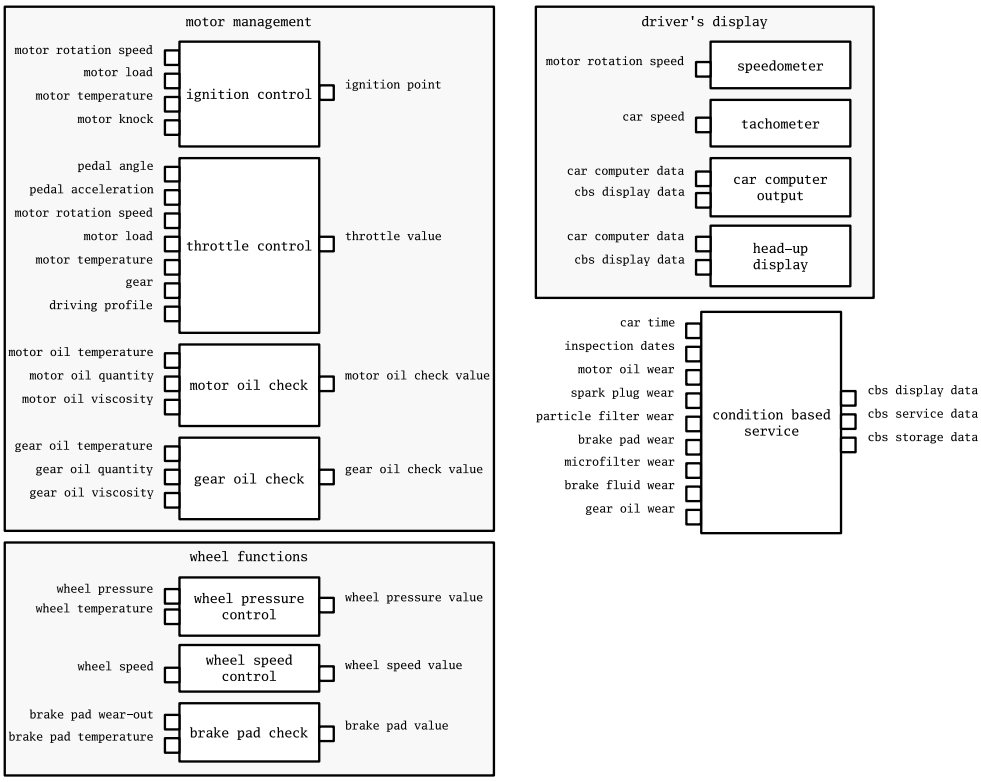
As can be seen, every functionality is arranged into the hierarchy (see Figure 2 for a graphical representation). Thus, four organizational units take care of the system: the motor unit, the MMI[5] unit, the CBS unit, and the wheel unit. The arrangement of the functionalities can be questioned as follows: the functionality "brake pad check" is arranged into "wheel functions", because it is a wheel function and therefore should be modeled and implemented by the wheel unit. On the other hand, the functionality "CBS" needs the values of "brake pad check" as well,

[5] man machine interface

therefore it could have been arranged into "CBS". A third possibility could have been, to subsume the functionality "CBS" under "wheel functions". This would have lead to problems with other functionalities, that affect "CBS" as well.

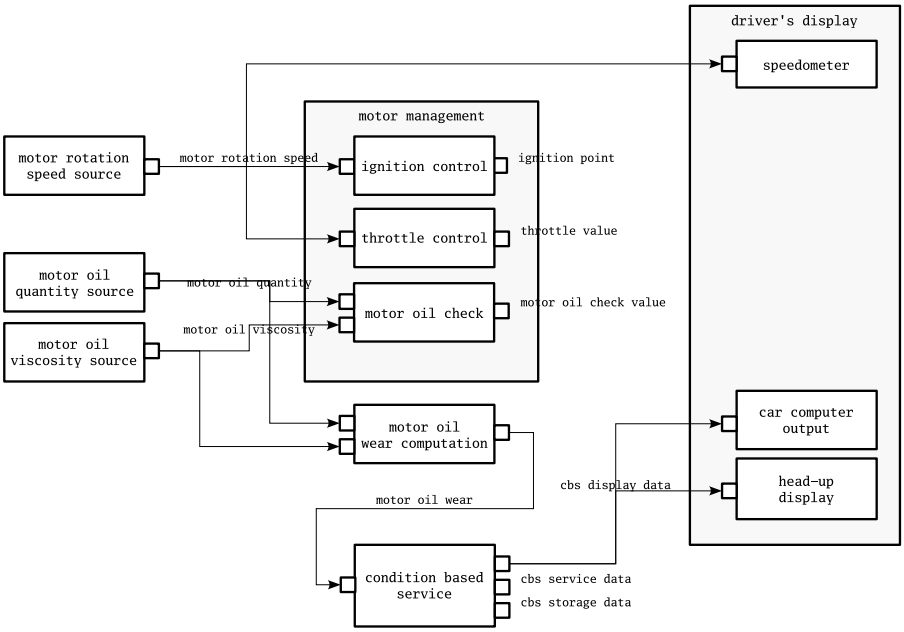
As we can see, even the first hierarchy contains functionalities, that cannot be arranged unambiguously. In order to clarify the problem, see Figure 3 for a enhanced architecture diagram, in which the required and provided information is shown. Here we can see in detail, that several of the information needed by the functions is provided by other functions or needed by other functions too. An example is the information *motor rotation speed*, that is needed by the functions *ignition control*, *throttle control*, and *speedometer* as well.

Figure 3 Hierarchy including required and provided information.



For the sake of brevity in the following we discuss only the signals *motor rotation speed*, *motor oil quantity*, and *motor oil viscosity*, the results are easily adoptable on the other signals and functions. In Figure 4 the relevant information sources and drains for the needed signals are modeled. It becomes clear, that several sources are used by several functions, which is a problem for scoping, as we will see.

Figure 4 Cutout of the system with information sources and drains.



2.4 Scoping

The problem is now to define system boundaries, scopes for the different functionalities. These scopes define, who is responsible for the design and implementation of a function. This supports the distribution of the development process, because every implementation unit can work independently. Therefore, the scoping should be as good as possible by means of minimizing the communication overhead and maximizing the cohesion of the components.

In the previous step we focused on three major units of responsibility, building the system's hierarchy: the motor management subsystem, the driver's display subsystem, and the condition based service subsystem. Every subsystem will be implemented by a separate unit. In Figure 4 not every function is assigned to one subsystem, these remaining functions have to be put into one of these subsystems.

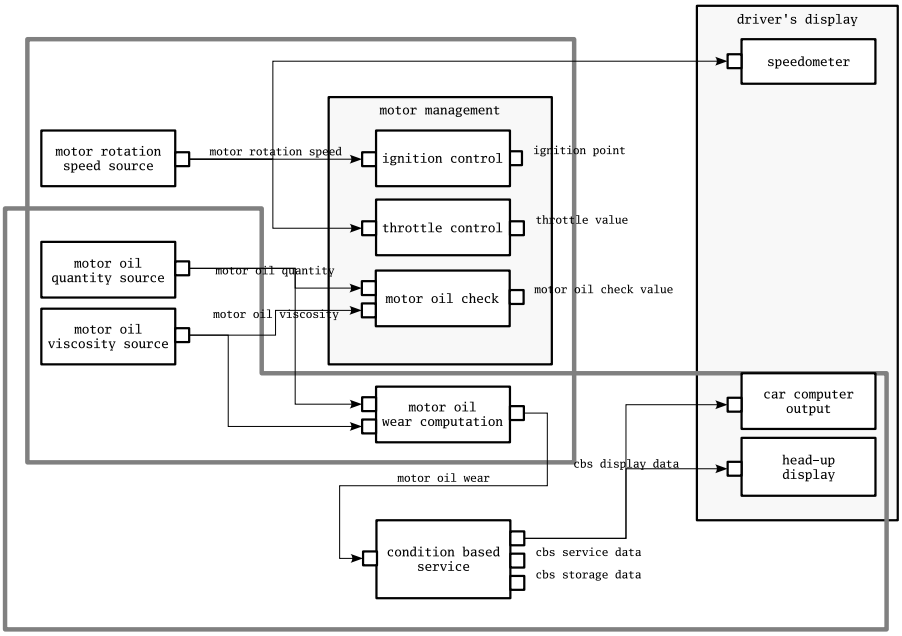
The task of scoping deals with these functions. All functions have to be assigned to a subsystem, in other words, the scope of the subsystems has to be set in a way, that the subsystems are disjoint and every function is assigned to one subsystem. This allocation of functions is not unique, but can be done in different ways. The function *motor rotation speed source* delivers data to *motor management* and

driver's display as well. Therefore it could belong to either scope. In this case, one could argue, that the function "naturally" belongs to *motor management*.

But the function *motor oil wear-out computation* is a function, that is only necessary for *condition based service*. On the other hand, the knowledge to compute the data is located within the *motor management* unit. Therefore, it should be located in the according subsystem. The problem continues, if the providing source and drain functions are viewed, they too have to be placed into one subsystem.

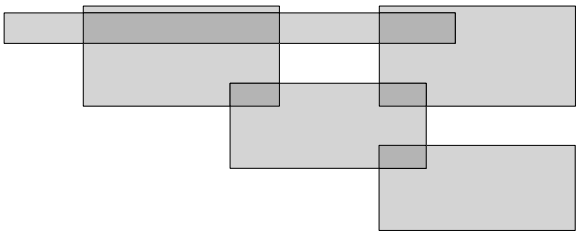
In short: the scopes of the subsystems overlap. Figure 5 shows a selection of possible, overlapping scopes of the functions.

Figure 5 Possible scopes of the system (cutout).



Abstracting the problem leads to the general problem of overlapping concerns. Whether the concern may be organizational units, functionality, or another criterion, one almost always gets overlaps, because of the nature of the problems, that is not disjoint. Figure 6 shows the abstraction, as can be seen, the scopes of the decomposed concerns overlap, they are intertwined.

Figure 6 Abstract visualization of overlapping scopes/concerns.



2.5 Systematization of the problems

In order to systematically present and discuss the problem mentioned before, the problems are described systematically in this section. First we will abstract the problem for the sake of clarity. Secondly, we will discuss the effects of the problem in three ways: component scattering, component tangling, and component crosscutting.

Overlapping components

The problem of scoping bases in the impossibility to uniquely separate components (functions) according to one decomposition criterion. This problem is well known in literature, it is often referred to as the "tyranny of the dominant decomposition"(e.g. [IBM, OT99]). It says, that the decomposition along one concern is not sufficient for other concerns, that would be cut into several items by this decomposition. In general, concerns overlap, thus the resulting decomposition should overlap as well. Solutions for this problem exist, e.g. in software development IBM introduced Hyperspace as solution [OT99], AspectJ introduced aspect-oriented programming [Asp].

In the following a short categorization of the problems caused by overlapping concerns is performed. The main problems are pointed out, so the existing solutions and the new approach can be compared by means of providing a solution to the problems. First, we will describe the effect of *scattering*, after that, the effect of *tangling*, followed by *crosscutting*. In the following we concentrate on the description of the problem, after that we describe the solutions and approaches so far.

Scattering

Scattering of a concern means, that the component, describing the concern, is used in several other components. This effect can be seen, if a component could be placed hierarchically in several components. In order to solve this problem, the developer decides to include the same component in all related components, therefore duplicating the component for every use. In programming, this is the typical copy-and-paste-problem, where the scattered code is hard to maintain, side effects or inconsistencies have to be dealt with.

An example is a logging functionality, that has to be integrated in every security-relevant function of the system. Often, the logging functionality is copied into every function concerned, thus scattering the logging functionality.

Tangling

Tangling of concerns means, that several concerns are implemented within one component. This effects occurs, if a component is affected by several concerns, and, in order to disjoin the component from other components, all concerns are implemented in this component. The result, implementations in one component, that belong to different concerns, leads again to components that are hard to maintain, because of its inner complexity and the missing separation of concern code.

An example is the *motor oil quantity source*. This function has to provide the motor oil quantity for the motor management concern. It too has to provide the wear-out of the motor oil in percent for the CBS concern. If both functionalities are implemented in one function, this function contains tangled code.

Crosscutting

Crosscutting of a concern means, that the concern spans multiple units of decomposition. Therefore, in order to disjoin these units, concerns have to be cut at the borders of the decomposition units. This leads to pieces of a concern, that are spread throughout other concerns, therefore the concerns are crosscut.

An example is the concern *condition based service*. For example, the function *motor oil quantity source* could be placed into *motor management*, providing an providing interface for CBS. Thus, the concern CBS crosscuts the concern motor management.

3 Related solutions and work

In the following related solutions and work are discussed, that deal with the problem of overlapping concerns. We will show, how the problem is tackled, and which advantages or shortcomings exist. First, component-oriented methods are discussed. Secondly, the idea of secondary hierarchies is presented. Lastly, aspect-oriented techniques are discussed.

The presented solutions are no complete list of existing solutions to the problem. The selection shows the range of pragmatic approaches (component-oriented) over modifications of existing approaches (secondary hierarchies) up to new approaches (aspect-orientation).

3.1 Component-oriented cuts

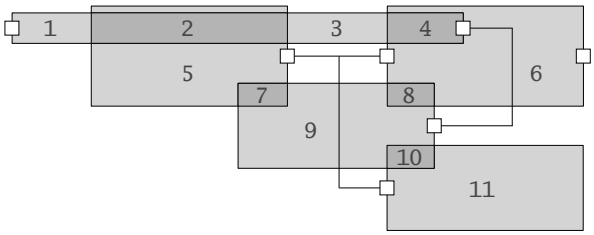
Component orientation demands for disjoint components, no overlapping components are allowed. This leads to well separated components that communicate through well-defined interfaces. Component orientation has vast advantages in programming and modeling when it comes to flexibility, reusability, modularization, encapsulation, etc. Therefore, component orientation should not be rejected in this context. As can be seen further, all other approaches, including our new approach, base on component orientation, they mainly enhance the component-oriented methods with new possibilities to use overlapping concerns in order to construct component architectures.

Having said this, how do component-oriented approaches solve the problem of overlapping concerns? Generally speaking, they solve the problem by ignoring it. By "ignoring" I mean, that the problem of overlaps is solved by defining interfaces for the overlapping sections, thus cutting them in several components; without regard to keep the information of the overlaps. The concrete strategy depends on the circumstances of the development.

In Figure 7 the example of Figure 6 is visualized for components. The components overlap, this situation is not allowed in component architectures. Therefore the components have to be disjoint.

Figure 8 shows the easiest solution of the problem, every overlapping section is cut and converted into an interface. This simple approach generates disjoint

Figure 7 Abstract visualization of overlapping components.



components, that tend to be small but specialized. Therefore, following steps in real development concentrate on hierarchic reorganization of the components in order to reduce connections, interfaces, or number of components.

Figure 8 Abstract visualization of disjoint components.

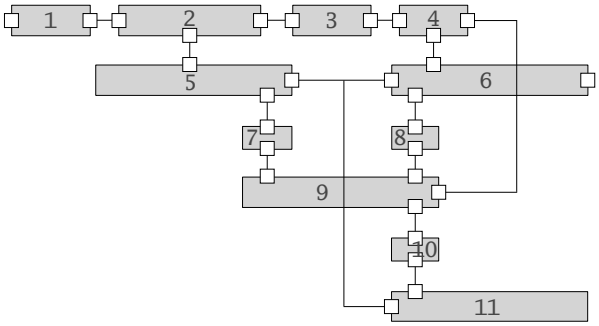


Figure 9 shows the result of a hierarchic reorganization of Figure 8. The overlaps of Figure 7, that suggest a certain hierarchy, were used for structuring.

This leads to a better structured, maintainable model. Figure 10 shows the advantages of information hiding concerning readability and complexity of the model. In the figure, the inner components are hidden, the main structure and information flow is visible.

As can be seen, the aforementioned effects of *tangling* and *crosscutting* appear. Depending on the concrete system, *scattering* can occur too. Even in the abstract example, the cut component parts, that belong to several components, are in a hierarchy relation with one unique component. Therefore the components crosscut each other with this component part.

In Figure 4 we showed a cutout of the CBS example system with information

Figure 9 Abstract visualization of restructured disjoint components.

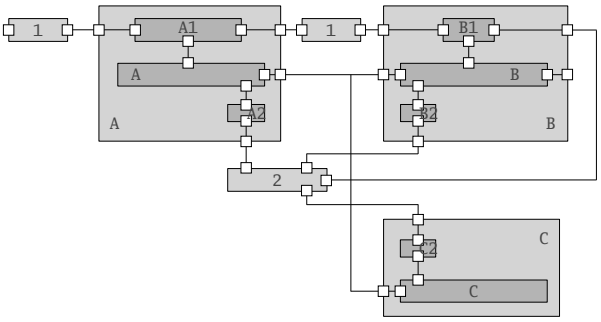
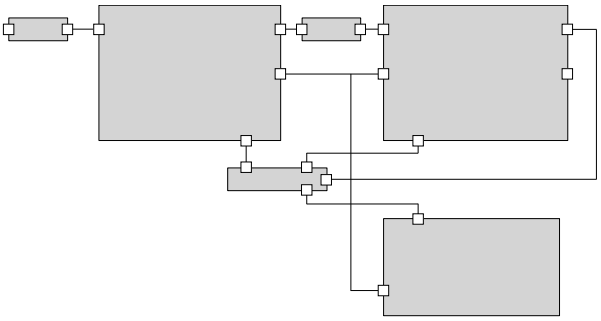


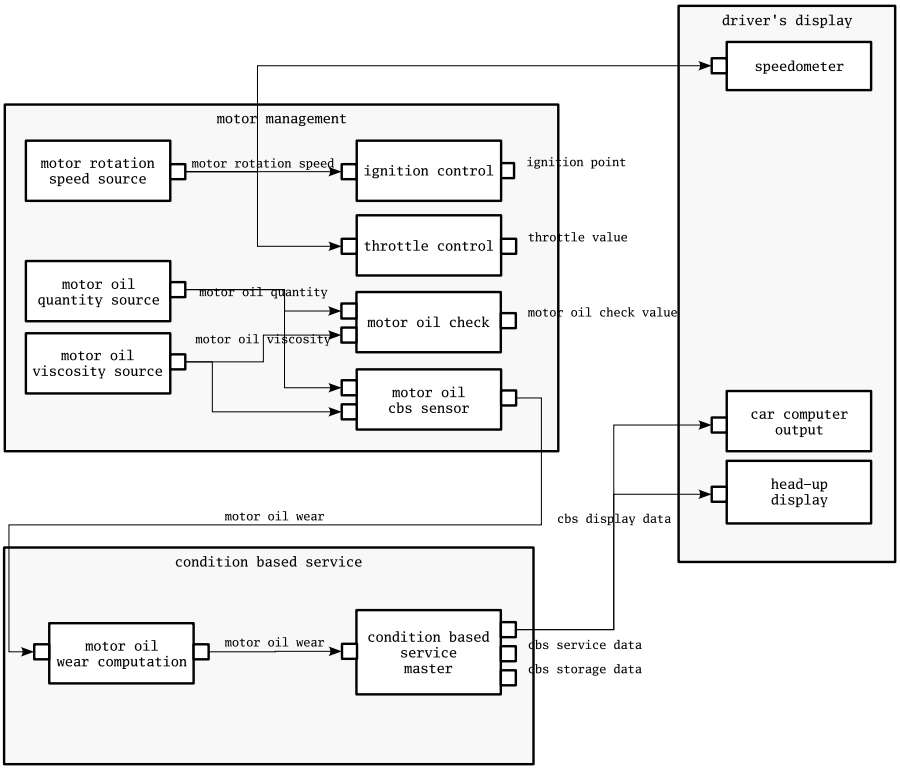
Figure 10 Abstract visualization of restructured disjoint components (hidden inner functions).



sources and drains. Based of this system we will illustrate the component-oriented approach further. In order to match the CBS example given in [Gro08], one change had to be introduced: The function *motor oil wear-out computation* was splitted into the functions *motor oil wear-out sensor* and *motor oil wear-out computation*. Figure 11 shows one hierarchy, that matches the functional and organizational needs. The functions are hierarchically allocated in *motor management* or *condition based service*, respectively. Therefore, *condition based service* was enlarged by a computation function and a master function.

In the method paper [Gro08] the scope of CBS is defined in figure 7 ("CBS system boundary"). In that figure, the computation is encapsulated by the *motor oil wear-out sensor*, and is thus not part of the CBS but of the motor management. This scoping makes sense, because the knowledge, how to compute the sensor data needed for CBS is located in the motor management unit. Therefore, the CBS sensor should belong to the *motor management* function. On the other hand, the *motor oil wear-out sensor* is a function of the CBS, meaning it is only needed

Figure 11 Exemplary hierarchy of the CBS cutout.



by CBS, not by any other function. Thus, we have the typical effect of crosscutting, and tangling.

The same argumentation goes for the display functions, e.g. *head-up display*, that have to display the CBS data. The requirements of what and when to display are set up in the context of CBS, therefore the display functions belong to CBS. On the other hand, the display functions have to handle more data than just CBS, therefore they create an own concern. Again, crosscutting concerns arise.

The display function *head-up display* additionally shows the effect of tangling, because the display of several data is managed by one function. Therefore the data of several concerns are handled within one component, which is the effect of tangling.

In this example, the effect of scattering is not present. This is because of the size and abstraction level of the example. Scattering arises to a greater extent when several other design steps are made, such as allocation of functions on hardware.

For the sake of brevity, the effect is not shown at this place.

Conclusion

As could be seen, component-oriented modeling gives the possibility to solve the problem of overlapping concerns. On the downside, the side effects of scattering, tangling, and crosscutting are introduced into the system model. Additionally, the overlapping concerns as information are lost, only the cut components are stored within the architecture.

3.2 Secondary hierarchies

The problem of overlapping concerns is a basic problem of component-oriented modeling languages. Therefore, when defining the architecture description language MOSES ([Kle06]), the problem was present too. At this time we decided to introduce "secondary hierarchies" into the modeling process.

For this concept, one defines a system architecture using a component-oriented method, such as MOSES. This system architecture is referred to as "primary architecture" or "primary hierarchy".

Now, the overlapping concerns are identified within this architecture, ignoring composed component boundaries etc. The functions, belonging to a specific concern are referred to as "secondary hierarchy", meaning a second, not fully expressive architecture.

This secondary architecture is defined as view on the primary architecture. Therefore, component boundaries of the primary architecture can be ignored, they even can be deleted, resulting in a secondary architecture with its own hierarchy. The secondary architecture allows viewing connections or describing conditions for the system. On the other hand, only a few modeling steps are allowed within secondary hierarchies, in order to avoid modeling inconsistencies within the primary or other secondary architectures.

Conclusion

Secondary hierarchies allow the modeling of overlapping concerns. The use of these hierarchies is limited in expression potential as well as restructuring possibilities. Therefore they enhance component-oriented techniques but do not provide an equal development of overlapping concerns.

3.3 Aspect-orientation

Aspect-orientation was invented to solve the problem of overlapping concerns. It was described and used in AspectJ ([Asp]), and after that adopted for nearly all areas of programming and modeling. The approach we introduce in the next section is based on aspect-oriented principles. Therefore, the main issues and terms of aspect-orientation are described here. Additionally, a short overview over existing approaches of aspect-oriented modeling of component architectures is given.

3.3.1 Introduction

Aspect-oriented approaches focus on native description of overlapping concerns. Therefore, the different concerns are separated, and later woven into a whole system description. This leads to an improved modularization of the overlapping concerns, and thus, to a more understandable and maintainable model.

The different concerns are described using the modeling units: classes, modules, components, etc. After that, the concerns are related to each other, i.e. additional functionality of one concern is related to the original functionality of another concerns. After that, an automatic weaver can compute the resulting system description, containing the original and the additional functionality as well.

In order to fulfill these demands, several new concepts have to be introduced: point-cuts, advices, aspects, join-points, and aspect weaving. Every aspect-oriented language has to realize these concepts.

The advantage is the possibility to model aspects (overlapping concerns) natively and thus maintain the complete model information. On the other hand, more models have to be maintained, and only after the weaving process, the complete model can be seen. The concrete advantages and disadvantages of our approach are discussed in Section 5.

In the following, a brief description of the main terms of aspect-orientation is given. In order to illustrate the terms, we use AspectJ as example.

Point-cuts

First, the designer must have the opportunity to declare, where an additional functionality can be inserted. The aspect-oriented language therefore has to

provide means for describing this place. The definition of *where* to introduce the additional functionality, is called point-cut.

Aspect-oriented languages differ in the power of the definition of point-cuts. AspectJ only allows point-cuts before or after function calls. Other languages allow direct manipulation of code, or less flexibility.

Advices

An advice is the definition of *what* the additional functionality is.

In AspectJ an advice is the code, that is inserted at each point-cut. The code can change local variables, therefore, the additional functionality can alter the original program execution. The alteration can be forbidden in other languages, the impact of advices is language-dependent.

Aspects

An aspect is the combination of a point-cut and an advice. An aspect therefore is the additional functionality, that has to be inserted, and the place, where to insert the functionality.

An aspect of AspectJ consists of the additional code and the place, where the code has to be inserted.

Join-points

Join-Points are the set of possible targets, that can be defined as point-cuts. A point-cut is an element of the join-points. The join-points are limited by the power of the point-cut description. The more flexible this definition is, the more extensive the set of join-points get.

AspectJ only creates a small set of join-points, increasing the readability and limiting the side effects. On the other hand, the modeling possibilities are limited as well. Other languages allow more or less flexibility.

Aspect weaving

Aspect weaving defines the process of inserting advices into their point-cuts. Weavers in general look for point-cuts and execute the weaving algorithm, if they found a point-cut and an according advice. The point-cuts can relate to static characteristics, in this case, whenever a point-cut is found, the according advice is

processed. Point-cuts can relate to dynamic characteristics as well, e.g. assertions, or states of variables. In this case, the dynamic condition has to be fulfilled in order to start the weaving process. Additionally, the weaving can take place before, during, or after further processing, such as generation, compilation, or execution.

3.3.2 Effects

What effects does the use of aspect-oriented techniques have, concerning our view?

First, using aspect-orientation, one must differ between core functionalities and additional functionalities. Generally speaking, aspect-orientation provides the possibility to describe two aspects of a system independently, without having to care, how the aspects are related. This is not fully true, because for the definition of point-cuts, some information of the aspects have to be known, but in general, the modularization is improved and the interdependencies are reduced in comparison to component modeling.

Secondly, combination of aspects can be described. Depending on how the concrete language is formulated, aspects can be woven only into the base model, or into other aspects as well. Using the latter mechanism, combinations of aspects can be described as interdependent aspects. This allows separate modeling of dependent aspects and improves again the modularization and readability of system designs.

Thirdly, modularization and readability improve, but the understandability of the overall system often reduces. This is based on the fact, that only the woven system contains all information about the system itself. Therefore, in order to extract the system functionality, several aspects have to be looked at, that are only related to each other via point-cuts, that are not visible in the target architecture. This is a principle problem of modularization, but is amplified through the invisible declaration of point-cuts, viewed from the weaving target.

3.3.3 Aspect-oriented approaches in the literature

As mentioned before, aspect-oriented techniques spread over all concerns of system development. The main focus lies within the programming community, that started aspect-orientation with AspectJ. Another main focus is aspect-oriented

modeling, which concentrates on modeling aspect-oriented UML diagrams in order to generate code. In recent years, the early phases of system design have gotten more and more interest, starting with aspect-oriented requirements engineering up to aspect-oriented system design.

The latter is the most interesting focus for this paper, because modeling of component architectures lies within this segment of aspect-orientation. In the following, a brief presentation of related, or interesting work is given.

Generalizations

[KTG+06] compares aspect-oriented software architectures using seven important modeling concepts of aspect-orientation: aspects, components, point-cuts, advices, static and dynamic crosscutting, and both relations aspect-component, and aspect-aspect. From this comparison, an integrated AO approach is extracted, that generalizes the different principles. Furthermore, four guidelines are defined, that should help to model one's own aspect-oriented notation. Although the definitions aim at UML models, we will discuss our approach regarding [KTG+06] in Section 5.3.

An approach to integrate different aspect-oriented approaches into one universal approach is described in [SSK+06]. This results in the introduction of am AOM reference architecture. Our approach is set in relation to this work in Section 5.3.

Ideas for the systematization of the symmetry of aspect-oriented approaches can be found in [HOT03]. Our approach is classified in Section 5.3.

[MT97] defines a framework for classification and comparison of ADLs (architecture description languages). Although the work is relatively old, the definitions and classification parameters still remain up-to-date. The VEIA models in their whole align with the requirements for ADLs given in the paper, the comparison is not part of our paper.

[BCG+06] describes seven issues for architecture definition languages, that have to be answered or fulfilled in order to support modeling of aspects. The issues are good starting points for evaluating an approach. However, the result of the paper does not reflect our observations, Batista et. al. do not see the need for new architecture elements but connectors and configurations. As we will see, in our approach, the main component elements need to be extended too (see Section 5.3).

[NPMH02] discusses, which extensions have to be made to an architecture description language, in order to support aspect-oriented modeling. The issues are

regarded in our approach, as is shown briefly in Section 5.3.

Variance

In [MRW06] an approach is introduced, that describes variance with aspects. Every aspect is viewed as a concern, that has to be modeled. Thus, an unaltered base model can be modeled, that the variants are woven into. The paper discusses the advantages of this approach in terms of legacy tool support, and clearer definition and separation of base and aspect models. The problem of interdependencies is mentioned, but not solved. The paper describes some techniques and elements, that are used in this approach too, it focuses on variance, and does not separate aspect definition and aspect instantiation.

Weaving

[ABE02, Pre02] describe weaving of behavior descriptions with statecharts. These are examples for the approaches to weaving behavior. In the following, we presume weaving of behavior to be executable and, therefore, will not discuss it further. More general information about weaving of models can be found in [CvE07, KL05].

Notation/UML

There are numerous approaches, that describe aspects using UML, e.g. [BGL04, HTO02, Her02], to mention but a few. In these papers, the modeling of aspects is done by means of UML, in order to use or reuse existing tools for modeling of aspects. In our approach, notation is not the main focus, we first have to define the basic principles. After that, an appropiate tool has to be found and adapted.

Compatibility

[BPS06] discusses the determination of compatibility of embedded software components. The compatibility could help ensuring correct interfaces e.g. for the pattern part of the approach, introduced in the following. For the sake of brevity, this discussion will not be carried out in this paper.

Other languages

[dF07] discusses the introduction of aspect-oriented constructs in the Architecture Analysis and Design Language (AADL). Most of the problems and solutions described in the paper are similar to that of our approach. This includes problems

such as order of weaving, interdependencies, or consistence. Interestingly, most of these problems are open issues there as well.

[PFT05] describes a platform for component and aspect based software development. They introduce a new model, CAM (component and aspect model), with the according infrastructure DAOP (dynamic aspect-oriented platform). After a good comparison of existing approaches, they concentrate on the platform and the resulting join-point declaration. Though the work seems a similar approach, our approach aims at a more abstract level of modeling. An important similarity, nevertheless, consists in the treating of aspects and components as first-level elements.

[PSCD06] describes FAC (fractal aspect component), an aspect-oriented extension of *Fractal*. Their aim, too, is to model component-based and aspect-oriented systems. The focus of the paper lies in the different bindings of components and aspects and in the clarity of the used concepts. In focusing on *Fractal* and the issue of bindings, the approach is suited for generative models, but not equally suited for abstract scoping models, such as the VEIA function model.

[PRJ+03] describes a new language PRISMA for modeling aspect-oriented software architectures. They distinguish different kinds of aspects, that are modeled with their own constructs. The approach is far too detailed for our field of use. Furthermore, the distinction between different aspects and components complicates the use of the solution.

[GCB+06] discusses an extension of ACME with aspect features. The main point of the paper is the fact, that only one small extension is needed: an aspectual connector. All other aspect-features can be modeled with existing language features. Their approach, too, aims at seamless integration with as less extensions as possible. As said before, more extensions are needed in our approach, for the sake of modeling clarity and expression power.

Other uses of AO

[NPM05] introduces an approach for tracing changes in architectures via aspects. An aspect is declared as entity to encapsulate evolution of models. This scope of use is too narrow in our case, but the mechanisms of encapsulation and the steps in defining architectures meet our experience as well. On the other hand, they build up a special aspect architecture, which in our opinion is not needed.

[GBRS06] discusses the possibility to trace decisions throughout an architecture using aspects. The ideas are similar to the ones, that led to our approach. In contrast, their solution focuses on reasoning of aspects and on an understandable

definition, but only on modeling of concerns in the same modeling space. Furthermore, the need for multiple instances of an aspect is not mentioned. Therefore, although the approaches are very similar in their reasoning and solution principles, they differ vastly in the details.

Modularization

Several papers tackle the question, on how to modularize programs, or architectures best. An example is [SGS+05], that establishes design rules for modularization. In this paper, we take the different concerns, and therefore their modularization, for granted.

4 VEIA approach

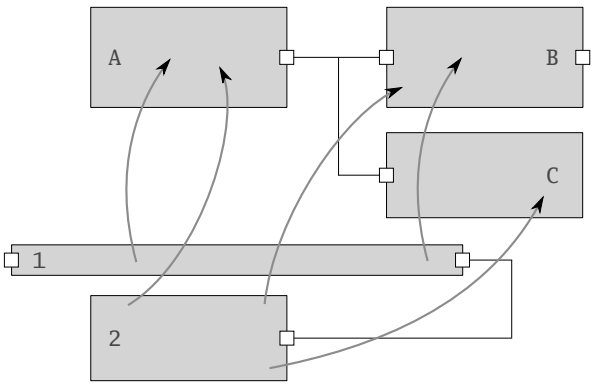
In this section, the VEIA approach for modeling overlapping concerns in a component architecture is presented. The approach is aspect-oriented and uses ideas of pattern engineering as well.

We first introduce the main idea of aspect-orientation using the abstract example of the preceding section. After that, the ideas are shown using the example of CBS. In this step, the concept is enhanced with the separation of aspect and instance models. Finally, the concept is formalized.

4.1 Aspect-oriented modeling

The problem shown so far is to model the overlapping concerns of Figure 7. The first idea of the approach is to model every concern as an independent model, as shown in Figure 12. The model of one concern is called *aspect model*. The overlapping parts are set in relation to each other using one of the new relations *inner*, *identity*, *copy*, or *replace*[6].

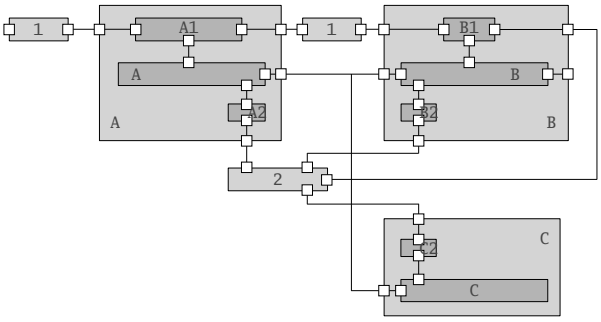
Figure 12 Abstract modeling of overlapping components.



[6] The relations are described in detail in the following section.

Thus, every concern is modeled completely as a component architecture and independently of the other ones. The modularization of concerns is increased. Now, an algorithm has to be defined, how to weave the different models into each other. Generally speaking, the related components are merged (*inner* relation) or replaced (*identity* relation). As a result, a conventional component architecture is created (Figure 13).

Figure 13 Abstract result of weaving.



The result does not differ from Figure 7, thus the expression power is not decreased. But in modeling separate concerns, i.e. in keeping the original concerns, the maintainability and understandability of the design increased, because the source model of Figure 12 is kept. Changes can be carried out in the aspect models, rather than the woven model, they can be located in the concern, they belong to.

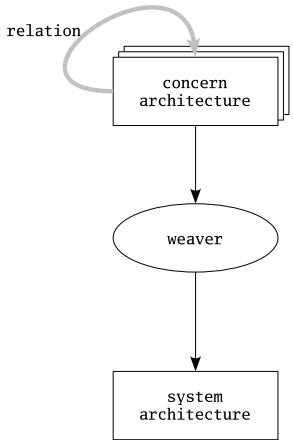
The chosen principle is shown in Figure 14. Every concern is modeled in its own architecture. The interdependencies of the concerns are modeled as relations. The weaver weaves all input architectures into one overall system architecture. The shown principle is a symmetric approach, i.e. all architectures are treated equally. Other approaches mark one architecture as base architecture, these approaches are asymmetrical.

After this short introduction of the basic idea, we take a look at the more realistic example of CBS.

4.2 Aspect-oriented modeling (case study CBS)

In Figure 11 three functions are modeled: *condition based service*, *motor management*, and *driver's display*. These functions are now modeled as separate concerns, first, without relations of the concerns. Figure 15 shows the resulting architecture.

Figure 14 Aspect-oriented principle (symmetric).



Every concern is modeled as a hierarchic component, that contains the subfunctions needed for the concern. Therefore, looking at the concern component reveals all functions and their information flow, that are needed in order to achieve the designated functionality. Thus, the modularization of the concerns is improved in comparison to Figure 11.

One additional modeling construct was introduced: *cardinality*. The cardinality indicates, how often the tagged element is allowed to occur in the woven architecture. For more concepts and further explanation, please see the following section.

The aspect components so far are the *advice*, that is introduced into the system. In order to completely describe an aspect-oriented approach, the target architecture, the point-cuts, and the weaving algorithm have to be specified.

The target architecture is the architecture, that the aspects are woven into. In our example only *condition based service* is a supporting function, that could be treated as an aspect function. The other functions, *motor management* and *driver's display* are equal functions in the meaning of providing basic functions. So we have several functions on par with each other, not one unique base function. Therefore, we design a *symmetric* approach, i.e. aspects can be woven not only into a base architecture, but into each other as well.

The advices have yet to be expanded by point-cuts, i.e. the targets, the aspects are woven into. The point-cuts are defined using relations between the elements of the architectures. We use two different relations: *identity* and *inner*.

Figure 15 Aspect-oriented model of the CBS concerns without relations.

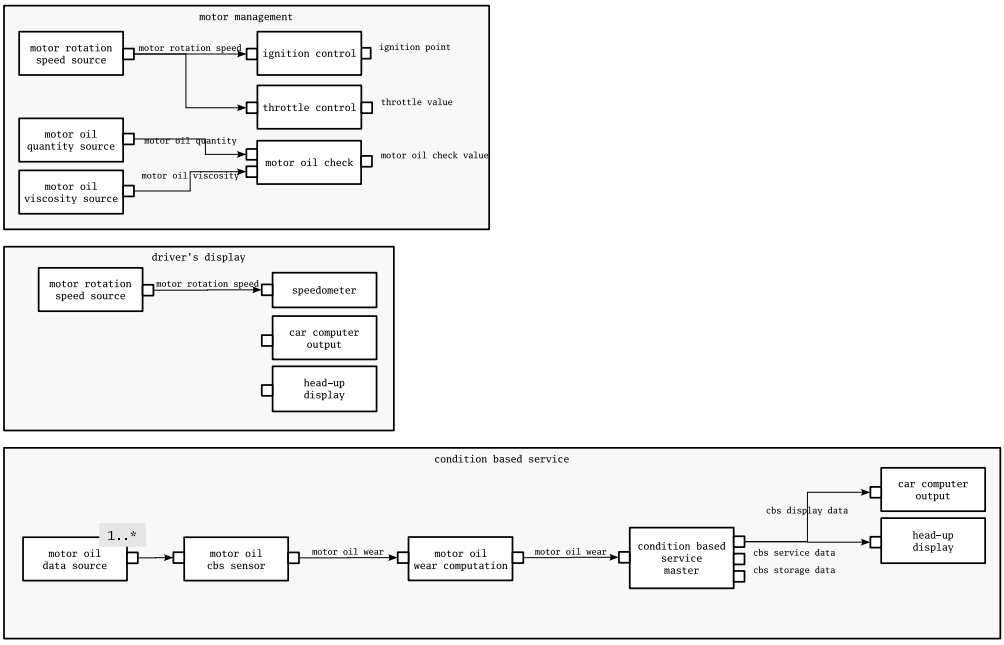
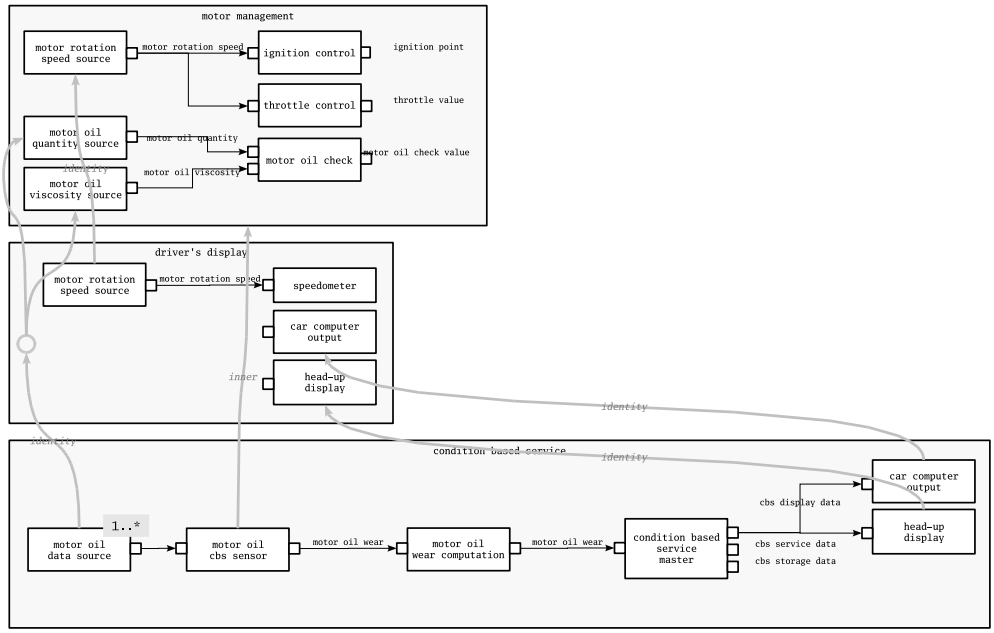


Figure 15 contains several redundant functions, e.g. the function *motor rotation speed source* occurs twice in the model. These redundancies are solved by the *identity* relation. This relation shows, that two functions are identical, i.e. the weaver leaves only the target function in the model, merges the source function into the target function, and rewires the according connections. In Figure 16 the aforementioned function *motor rotation speed source* of *driver's display* is in *identity* relation to the according function in *motor management*.

The *inner* relation demands that the source function is placed into the target function. If the target function would be an atomic function, the meaning of the relation would be unclear. Therefore, the target of an *inner* relation can only by a hierarchic component. In case of doubt, a new hierarchic component has to be modeled in the target architecture, or a higher hierarchic target has to be chosen. In Figure 16 the function *motor oil cbs sensor* is set in *inner* relation to *motor management*, therefore in the resulting function architecture the sensor will hierarchically belong to the *motor management* concern.

In this example so far the point-cuts are the functions of the target architecture. In addition, ports, signals, and connectors are point-cuts for the *identity* relation as well, this is not shown for the sake of brevity.

Figure 16 Aspect-oriented model of the CBS concerns with relations.



Finally, we need a weaving algorithm. In this document, the algorithm is described very briefly. Open issues, such as the order of the weaving steps etc. are mentioned in Section 5. At this point, the general idea of the algorithm should be made clear.

The weaving algorithm starts with two architectures, one target and one source architecture. The algorithm first processes the functions that are in *inner* relation. The source function is placed besides the child functions of the target functions. The connections of the source function are rewired, if needed, the according ports are created or deleted at the target function.

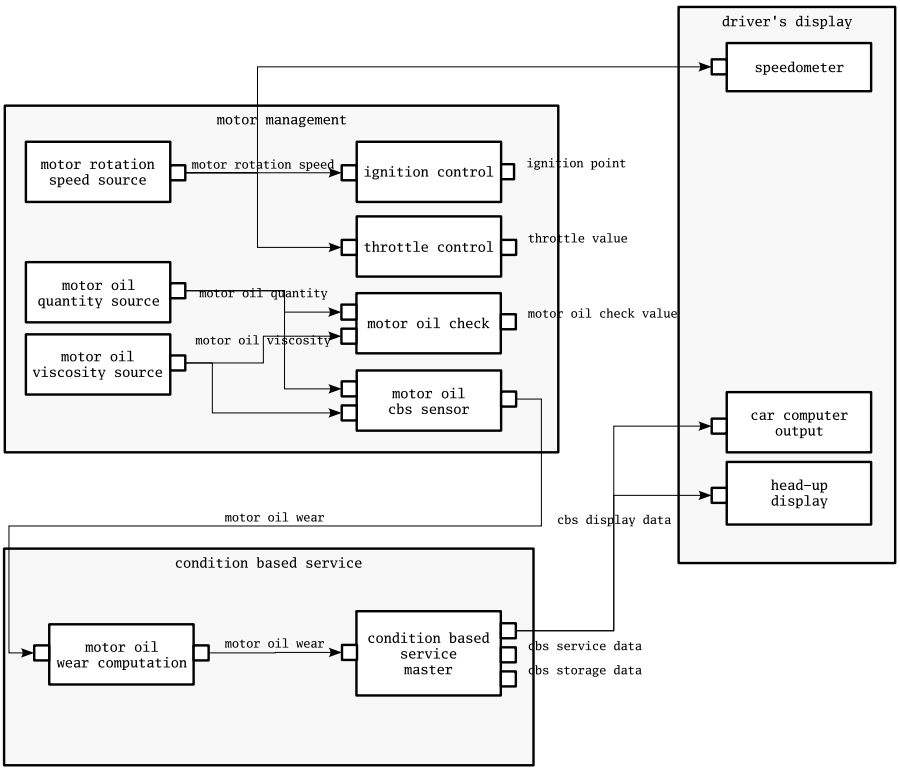
After processing the *inner* relations, the *identity* relations are processed. First, the ports and signals are unified in the target function. If the source function does not contain additional ports or signals, nothing happens, otherwise the additional ports or signals are created at the target function. The connections of the source function are rewired accordingly with the target function. If the source function does contain behavior, the behavior of the source and target functions are merged in the target function. Now, the source function is deleted.

All other functions are placed besides the topmost functions of the target architecture. The hierarchy of the source architecture is kept except for the functions, that were in a relation to the target architecture, they were placed outside the

original hierarchy in the previous steps. Now, the algorithm starts again with the next aspect architecture, until all aspects are processed.

Figure 17 shows the result of the weaving process with the architectures of Figure 16. The concerns are woven into one consistent architecture, just as expected. The approach so far is a straightforward aspect-oriented extension of the common VEIA models. In the following section, we will discuss more constructs and enhancements, that ease and extend the modeling possibilities.

Figure 17 Result of the weaving process.

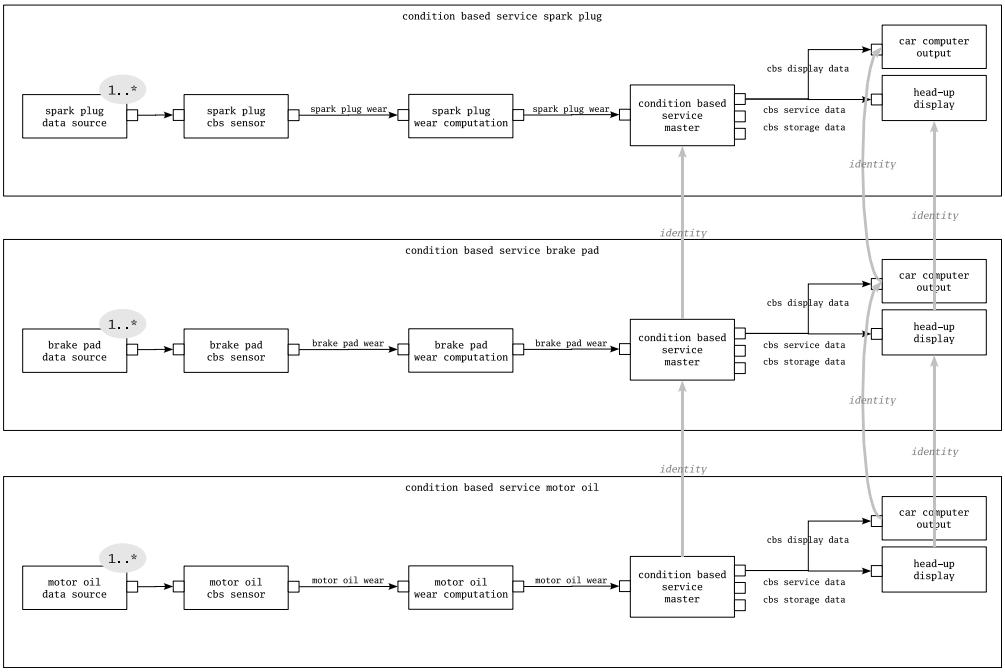


4.3 Aspect and instance models

After introducing the basic principle, that is used in our approach, we will now discuss the separation of aspect and instance models. This extension increases the modeling power by allowing native modeling of recurring model elements. When modeling aspects, one often notices, that several aspects are similar or share a

common pattern. We point this out using again the example condition based service CBS. Figure 18 shows the model of CBS for three volumes: *spark plugs*, *brake pad*, and *motor oil*. The models are very alike, differing in the sensor and computation functions. The *identity* relations indicate, that the master and display functions are equal.

Figure 18 Models for three volumes of the CBS concern.



Furthermore, if we decompose the computation function, as shown in [Gro08], we see a specific and a generic part. Figure 19 shows this decomposition. The left function *wear normalization* computes a relative wear in percent from the absolute wear value of the according sensor. This function differs in each CBS aspect. The right function *wear computation* computes the remaining time or distance, stores the value etc. This function is identical in each CBS aspect.

For the given aspect CBS we can identify identical and different parts. This is true, not only for CBS, but for many other concerns, such as logging, or error handling. This affects concerns, that have many instances in the architecture, that share a certain alikeness or even equal parts. It is possible to model these aspects with the given artifacts, but the modeling of one concern is again scattered. Therefore we will support the modeling of such concerns with a new construct: the separation of aspect and instance nets.

Figure 19 Decomposition of the CBS computation.

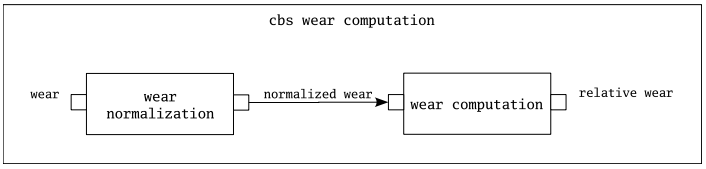
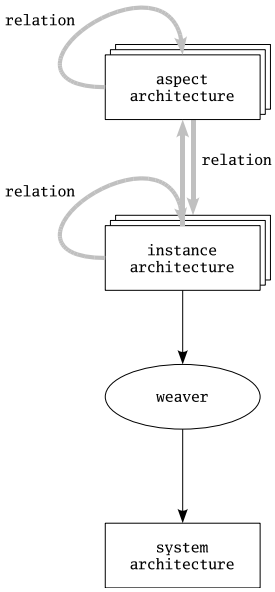


Figure 20 shows the new principle. A concern can be modeled as aspect architecture or instance architecture. The instance architectures are the architectures we used in the previous examples. The aspect architectures define a pattern for the instances, that have to be integrated into the system architecture. Thus a pattern for the aspect can be defined for the overall architecture. Please notice, that the instantiation of aspect architectures is not shown in the picture as arrow, for the sake of brevity.

Figure 20 Extended aspect-oriented principle (symmetric).

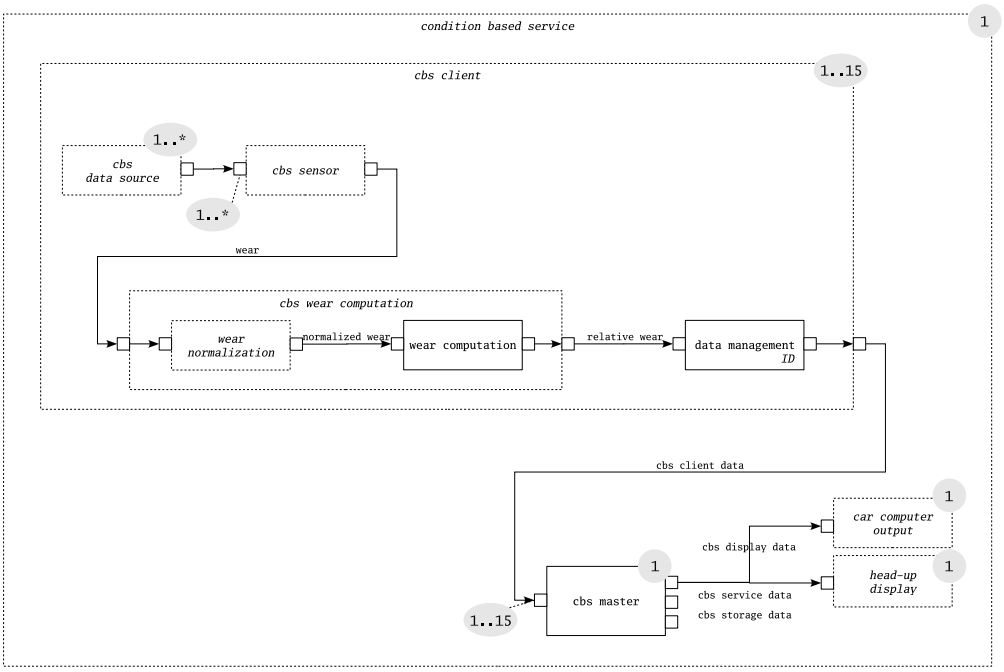


Aspect architectures

Now, we will define the aspect model for CBS. Note, that this aspect model is the basic model of the concern CBS, the single occurrences of the different volumes do not occur in this model. Figure 21 shows the aspect model of CBS. There are

several new constructs in the model: abstract functions, cardinalities, and abstract variables. The model and the constructs are discussed in the following paragraphs.

Figure 21 Aspect model for CBS.



The main function, that encloses the whole aspect, i.e. the root function of the architecture is *condition based service*. It is an abstract function, recognizable by the dashed border and the oblique text. Abstract functions are functions, that are not fully modeled, they can be changed during instantiation or during weaving. That means, either they contain hierarchically abstract functions, or they are not fully modeled or not modeled at all, concerning behavior, ports, and signals. Abstract functions are used, if a function has to be modeled in the instance architecture, but the implementation depends on the instance, i.e. it cannot be defined in the aspect architecture. Abstract functions can occur in instance architectures, too, as is explained further down in this paper. Signals can also be abstract, meaning that the signal is not fully defined, it can be defined during instantiation or weaving. Abstract signals are not modeled in the example.

The function *condition based service* is composed of four functions: *cbs client*, *cbs master*, and the two display functions. Every function has a cardinality, that determines, how many instances of the function can occur in the system architecture. The cardinality can be given as number, or range, with "$\star$" indicating no boundary. A cardinality can be defined for functions, and ports as well. So far,

cardinalities are independent of each other, i.e. in Figure 21 *cbs client* could be instantiated twice, whereas the according port at *cbs master* could be instantiated only once. The further refinement of cardinalities is an open issue of the approach, an outlook is given in Section 5.

In our example, the *cbs client* has to be instantiated at least once, it can occur in up to fifteen instances. For every instance a new port is created at the *cbs master*, therefore the port has a cardinality, too. The *cbs master* can only be instantiated once, thus every instance architecture contains the same *cbs master*. The same goes for the system architecture, it too has only one master function. The display functions are treated accordingly.

The function *data management* has the abstract variable *ID*. This variable shows, that a certain data element has a name, but is not fully defined yet. On the other hand, the whole function is completely defined, and therefore not abstract. Thus, a variable element can be introduced without opening the definition of a function. The modeling possibilities of the instance architecture are thus cut down to a controllable level.

The function *wear normalization* is an abstract function. It contains behavior, that is not shown in the figure, for the sake of brevity. Also, the input and output signals are defined. During the instantiation the predefined behavior can be modified, deleted, or kept as is, but the interface of the function must not be changed.

The functions with the most liberties are the *cbs sensor*, and *cbs data source*. Both contain no behavior, therefore the implementers are completely free. Only the provided part of the interface of *cbs sensor* is defined, and must therefore not be changed.

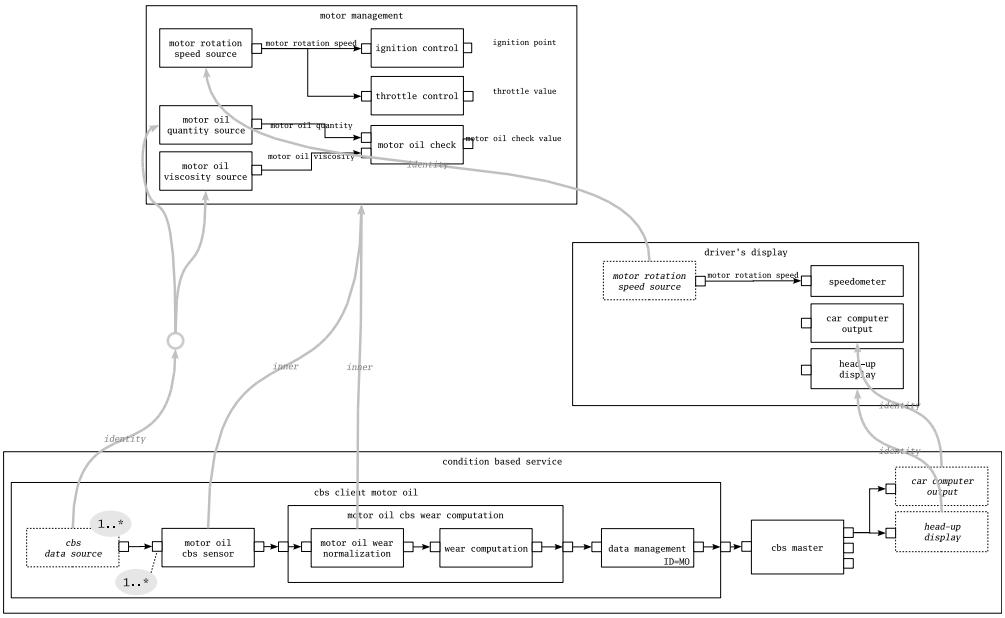
Altogether the aspect architecture allows the precise definition of a concern. There are defined ways of open declarations, that can be changed during instantiation. On the other hand, fixed elements can be defined, that must not be changed. Thus, the freedom of the implementors can be guaranteed without uncontrollable effects on the whole concern or resulting woven system architecture.

Instance architectures

After defining the aspect architectures, they are instantiated in order to create the instance architectures. This is done for every needed instance of the aspect, in the example of CBS an instance architecture is created for every volume that has to be controlled.

The instance architectures can contain, just as the aspect architectures, relations to other architectures. In the example of CBS the instance architectures are the first artifacts that define relations. Figure 22 shows the instance architecture of the CBS volume *motor oil*. The signal names of the instance architecture are omitted for the sake of brevity, in order to get the architecture drawn in one row.

Figure 22 Instance model for CBS volume motor oil.



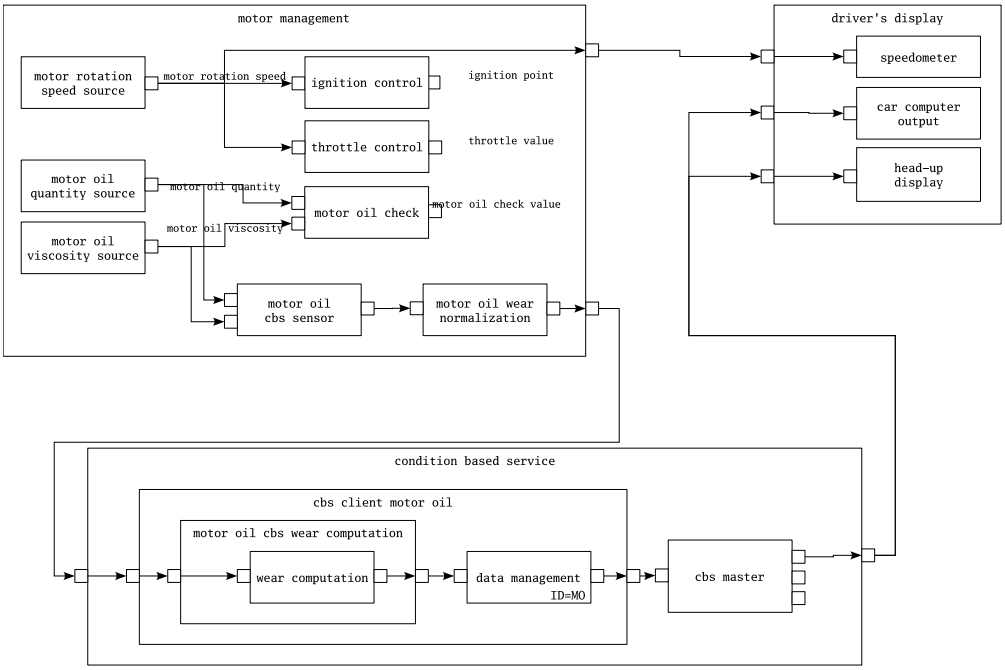
The instance architecture of Figure 22 contains the instantiated functions except for *cbs data source*, which is still abstract, because it is identified with the according *motor oil* functions. The same applies to the display functions. The instantiated functions are in relation to *motor management*. The abstract variable *ID* is instantiated with the value *MO*, that can e.g. be used in the behavior or in signals.

The instance architecture *driver's display* contains the abstract function *motor rotation speed source*. This architecture *driver's display* was created without an aspect architecture. This is allowed for architectures, that do not need the definition of an aspect architecture.

Weaving the given architectures results in the system architecture of Figure 23. Just as expected, the functions of the concerns are scattered throughout the system, but identifiable and manageable through the instance and aspect architectures. The functions are taken out of their hierarchy, so the *motor oil wear normalization*

can be positioned inside *motor management*. This leads to a hierarchic artifact like *motor oil wear computation*, that consist of only one function *wear computation*. This hierarchy could be removed automatically, at this point, we do not force such cosmetic corrections.

Figure 23 Woven system architecture with CBS volume motor oil.

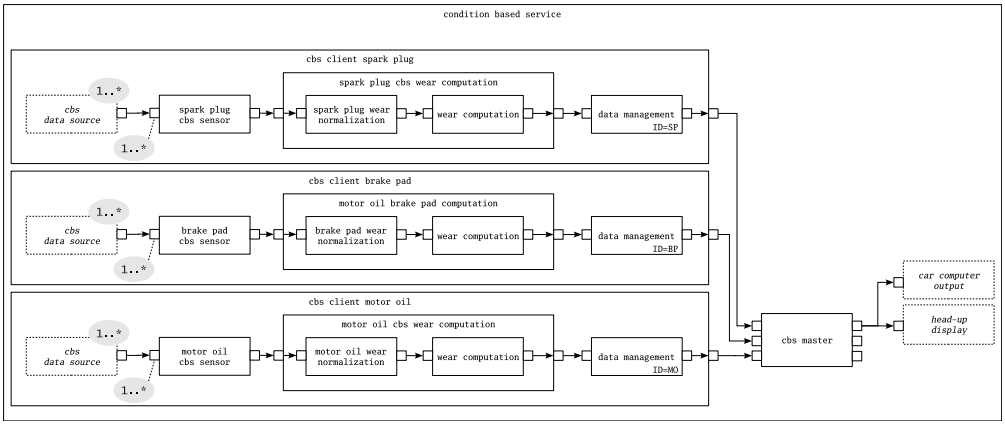


Turning to the remaining two volumes of the example, *spark plug* and *brake pad* we only show the CBS instance architecture for the sake of brevity in Figure 24. The point in this figure is, that the functions with cardinality "1" of Figure 21 occur only once in the instance architecture. If the instance architectures are drawn separately, these functions can occur twice or more often, then they have to be connected with an *identity* relation (not shown in the figure).

The woven architecture for all three volumes is not shown at this point, because of the size of the resulting architecture. The weaving does not differ from the examples shown so far. There are no new constructs, that have to be introduced with this example.

In this section, we introduced the separation of aspect and instance architectures and described it briefly. It provides means to separate the definition of aspects, that have to be used several times, from their instantiation without giving up the advantages of aspect-oriented techniques. In the next section we discuss patterns,

Figure 24 Instance model for three CBS volumes.



a special use of instance architectures. The formalization of the constructs is given in the next but one section.

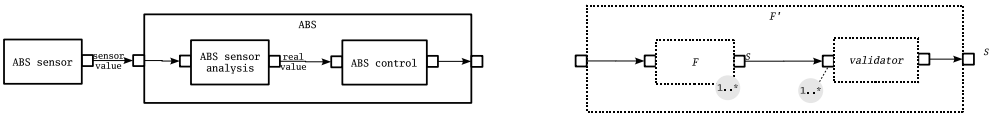
4.4 Patterns

There are several kinds of concerns, that can be integrated into an architecture. First, there are concerns that are developed as a new function architecture. These concerns are designed as instance architectures, that are woven. Secondly, there are concerns, that are used several times, they can contain variable and constant parts. These concerns are modeled with aspect architectures, their instantiation is woven into the system architecture.

This section describes a third kind of concerns: concerns that are described by a pattern. Patterns are used throughout several domains, in computer science the best known description is the description of design patterns by Gamma et al. [GHJV94]. The principle of design patterns lies in the description of general, reusable solutions as patterns, modeled in the used language. In our case, the patterns are described as aspect architectures. For each use of the pattern, an instance architecture is created. Now, each pattern can be associated with the elements it has to be applied to. Now, the weaver applies the pattern and creates thus the system architecture.

An example will show the use of patterns in function architectures. The pattern to apply is a simple redundancy pattern. The pattern is applied to the architecture of

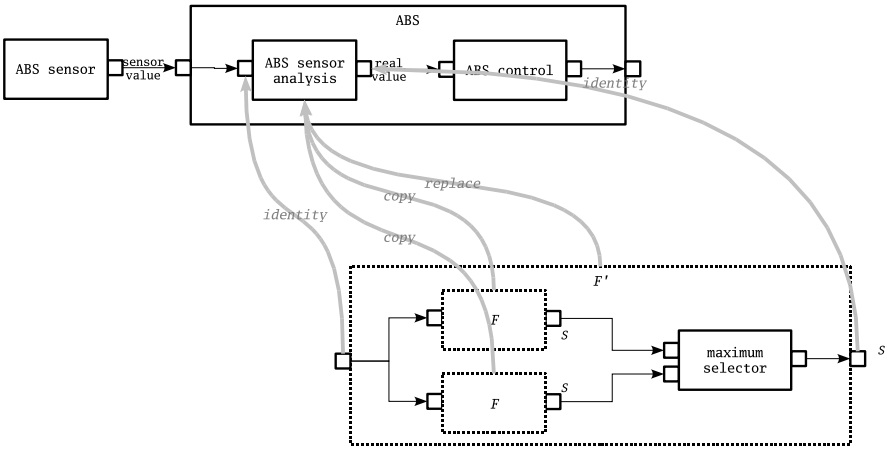
Figure 25 ABS and redundancy pattern.



a simple anti-lock braking system (ABS). Figure 25 shows the ABS architecture and the aspect architecture of the redundancy pattern. The pattern consists of one to many abstract functions *F*, that are connected to a *validator* function. The output signal *S* of the *F* functions is the same as the output signal of the whole function *F'*. The instance architecture will make the use and definition more clear.

Now, an instance architecture for a twofold redundancy is created and set in relation to the ABS architecture, both can be seen in Figure 26. In the instance architecture, function *F* occurs twice, the *validator* was instantiated to a *maximum selector*. It selects the maximum of the input signals *S* and forwards it to the output port. The relations of the instance architecture to the ABS architecture are as follows: First, the scope of application is set by the two limiting *identity* relations of the input and output ports. Secondly, both instances of *F* are copies of the function *ABS sensor analysis*, which is indicated by the *copy* relations. Lastly, the original function *ABS sensor analysis* is replaced by the instance architecture of *F'*, thus the *replace* relation.

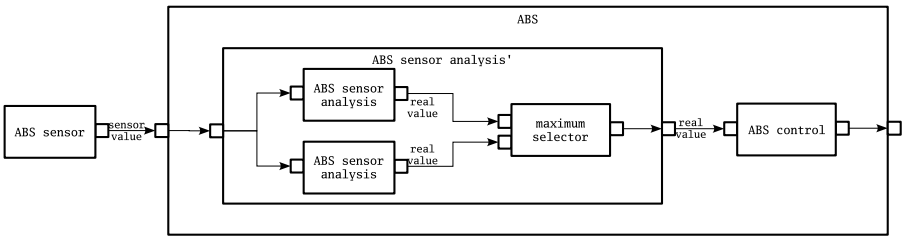
Figure 26 ABS and twofold redundancy instance.



In the description, a certain order of the steps is foreclosed. This is due to the fact, that the weaver first has to apply the *identity* and *copy* relations, after that, the

replace relations are evaluated. The result of the weaving process can be seen in Figure 27.

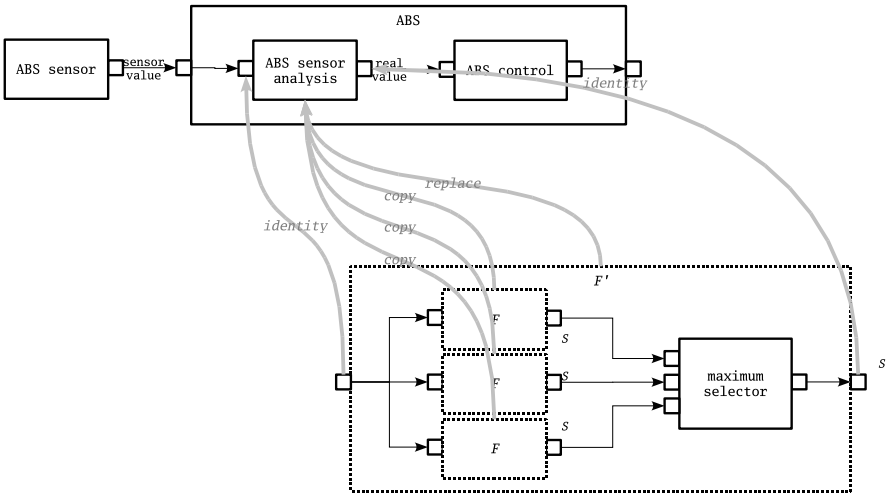
Figure 27 Weaving result for ABS with twofold redundancy.



As can be seen, the target function *ABS sensor analysis* is replaced by the pattern, the interface of the function remains the same, therefore the surrounding functions do not have to be changed. This is the only limitation of the usage of patterns: the interface of the target function need not to be changed. Otherwise the side effects are very hard to control. Research in softening this border remains to be done.

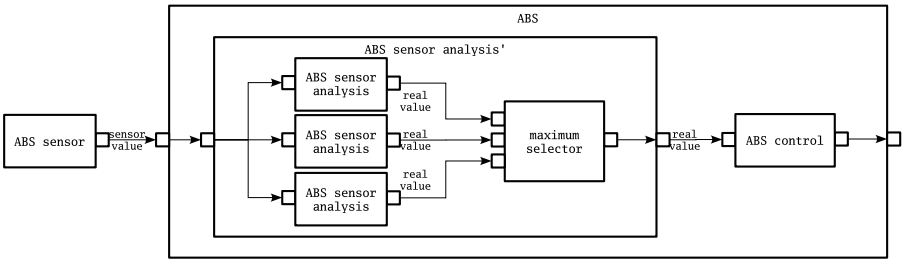
Figure 28 shows the instance architecture for a threefold redundancy instance. The resulting system architecture can be seen in Figure 29.

Figure 28 ABS and threefold redundancy instance.



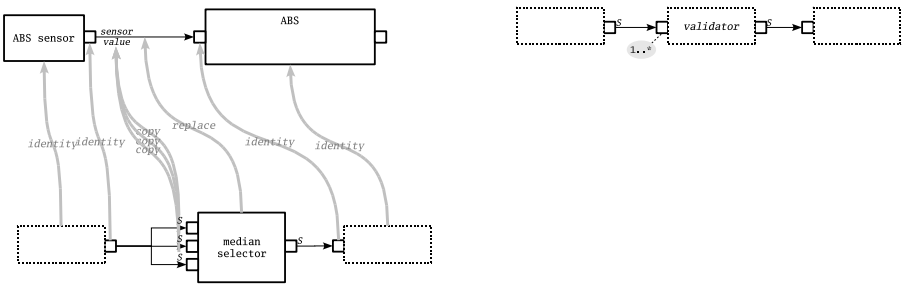
The use of patterns is not restricted to functions, but can be used with e.g. connectors, too. Another redundancy pattern will show this use case. The sensor

Figure 29 Weaving result for ABS with threefold redundancy.



data *sensor value* of the *ABS sensor* should be transferred redundantly in order to increase the safety of the system. The redundant transfer is done with different hardware connections, e.g. parallel wires. This redundancy is reflected in the function architecture, because of the function, that checks the different signals for consistency. Figure 30 shows the aspect architecture and the instance architecture for signal redundancy through connection redundancy.

Figure 30 ABS and signal redundancy (aspect and instance).

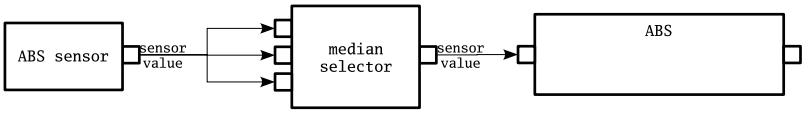


The aspect architecture consists of only the *validator* function, that is abstract, i.e. has to be defined throughout the instantiation. The input and output ports have the same signal, therefore applying the pattern again does not change the interface of the target element. The number of input signals ranges from one to as many as needed. The surrounding abstract functions and their ports define the context, or the scope of the pattern, respectively.

The instance architecture shows a threefold redundancy, that is resolved by a *median selector*. Again, the focus is defined using the *identity* relations of the ports, that are connected by the original connector. The according functions are identified as well, because of clearly setting the context. The signal *S* is copied three times, whereas the *validator* function was implemented as *median selector*.

The *median selector*, its ports, and the corresponding connectors replace the original connector. The result of the weaving step is shown in Figure 31.

Figure 31 Weaving result for ABS with signal redundancy.



Summarizing, aspect and instance architectures open modeling possibilities, that can be used e.g. in the definition of architecture patterns. Other usage surely is possible, but is not discussed in this paper. All uses have in common, that the aspect and instance architectures are kept throughout the development process, therefore the design decisions are preserved for later use, reuse, or change.

4.5 Formalization

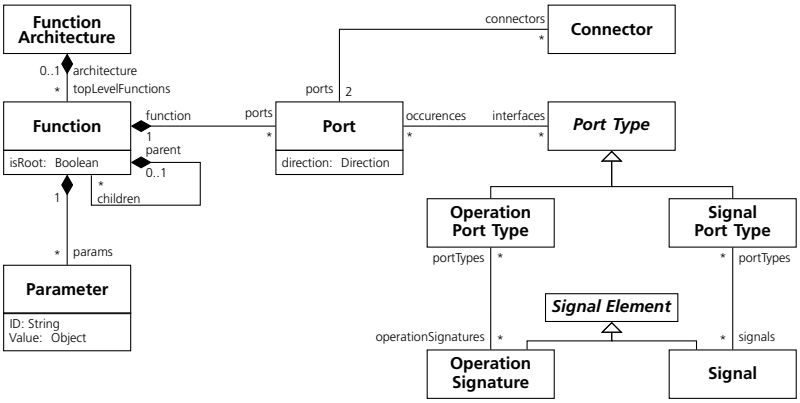
In this section the constructs are formalized in order to define requirements for an implementation of the concepts. The formalization contains the structural description of the changes in the metamodel. It also contains a semi-formal description of the weaving process in pseudocode.

4.5.1 Metamodel

Figure 32 shows the metamodel for function architectures without any extensions. This metamodel bases on the function models of VEIA. It is simplified for this document in order to show the basic principles of extension, that concern all metamodels, regardless of the underlying language. Therefore, the aspect-oriented enhancements can be adopted to other languages more easily.

A *Function Architecture* consists of *Functions*. The functions are in *topLevelFunctions* relation to the architecture. They belong exactly to one architecture, whereas an architecture can contain several top level functions. As one architecture does contain only one root function, this function is declared with the attribute *isRoot*. A function can be hierarchic, in this case it contains several other functions in the *parent–children* relation. A function, too, can be assigned parameters, every parameter has an *ID*.

Figure 32 Metamodel for function architectures.



Every function contains *Ports*, the ports are assigned to exactly one function. Every port has a *direction*, in VEIA the directions *in*, *out*, and *inout* are provided. The direction of the ports is indicated with the arrow of the connection in the figures of this document. This graphical notation can be freely chosen.

Ports are connected with *Connectors*. Every connector connects exactly two ports, but ports can be connected by several connectors. Thus, 1:3 connections as in Figure 31 are three connectors in the model.

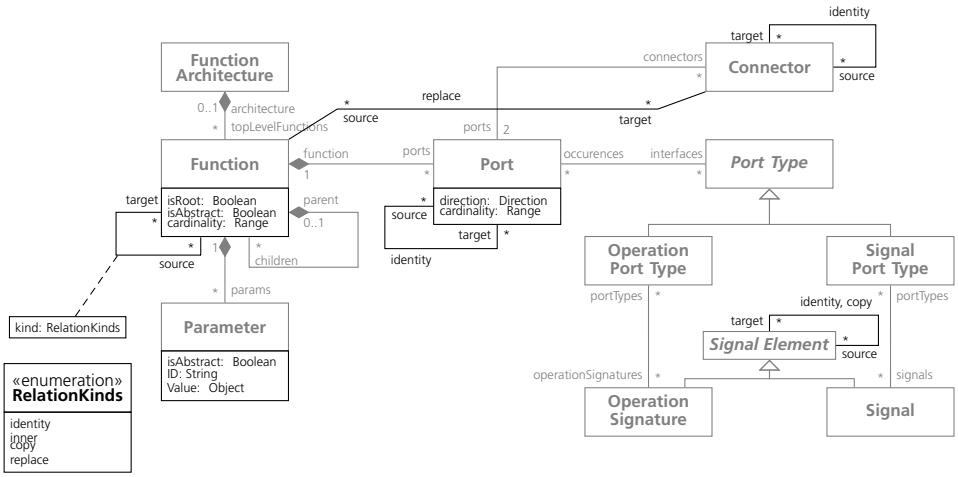
Signals are modeled as parts of *Ports Types*, the same goes for *Operation Signatures*. The port types work as interfaces for ports. This allows reuse of interfaces, signals, and operations in a model.

The extensions for the new approach are shown in Figure 33. The contents of the original model are grayed out, so the new elements can be identified more easily. There is one new class and several new attributes and relations, that are described briefly in the following.

The main class, *Function* has several changes. First, two new attributes are defined: *isAbstract* that indicates if a function is abstract, and *cardinality* that contains the cardinality of the functions. Secondly, a function can now contain abstract parameters using the *isAbstract* attribute of *Parameter*. Thirdly, the aspect relations *identity*, *inner*, *copy*, and *replace* are annotated to the relation of functions to functions. Lastly, a *replace* relation can be drawn from *Function* to *Connector*.

Three more *identity* relations are defined for *Port*, *Connector*, and *Signal Element*. *Signal Elements* can be in *copy* relation as well. Thus, the enhancements of

Figure 33 Extended metamodel for aspect description.



aspect-oriented modeling can be applied to the original model with only a few changes.

4.5.2 Weaver

The weaver carries out the model weaving following the weaving algorithm. The weaver itself can be a program, however, the weaving can be carried out by hand, too. In this section, we define the algorithm as a basis for further research or implementation. The automation or implementation of the weaving process and the algorithm is no goal of this paper.

The weaving algorithm is shown in Listing 1. Starting with an empty system architecture, all function or instance architectures are added, until the system architecture is created. The result is a hierarchic system architecture with all functions, ports, connections etc. woven into it.

Listing 1 Weaving algorithm (Pseudocode).

```
let system_net = empty architecture;
for every function or instance architecture fa do
begin
  for every element e in fa do
  begin
    if e has relation r
    then
```

```
      case r
        "inner"
          put e into target function
          create new ports at bounding function
          connect according ports
        "identity"
          merge e with target element (structure and behavior)
          create needed ports
          connect according ports
        "copy"
          replace e with a copy of target element
        "replace"
          after resolving all "copy" relations:
            replace target element with e
            connect according ports
      end
    else
      put e into system net (keep hierarchy)
      connect according ports
    end
  end
end
```

Of course, the algorithm has to be refined for practical use. For instance, the meaning of “replacing an element” has to be defined. These details are not tackled in this paper.

5 Discussion

In this document we presented a new approach for modeling systems with overlapping concerns. We described the approach and started its formalization concerning structure and basic weaving. The details of the approach were not fully covered, first, for the sake of brevity, second for some of them are open issues yet. In this section we will discuss the approach and the open questions.

5.1 Overall discussion

The approach is an extension of the existing VEIA models. Because of the simplicity and concepts of the VEIA models, the approach is an extension for component models in general. This means, the use of the approach is not restricted to a certain ADL, but can be applied to all ADLs, that contain the concept of hierarchic components with well defined interfaces.

The approach is a symmetric, aspect-oriented approach. This means, functions architectures, aspect architectures and instance architectures are equally important architectures concerning weaving one into each other. This eases the definition of aspects, because function architectures can be easily extended into first instance architectures. Whether the instance architectures should be abstracted into aspect architectures lies in the hands of the user.

This approach is defined and investigated for product architectures. An extension of the approach to meet the needs of product-line modeling was not focus of the research so far. Such an extension has to handle a new level of concerns, describing the variance of the product-line. As described in [MRW06], variance can be described with aspect-oriented approaches, too. In VEIA, we use a central product line description, and a local description of the variance in each architecture. Both approaches are equally powerful, they only differ in the metamodel constructs needed for the description of variance. In conclusion, the use of our approach in product-line architectures has yet to be investigated with all its consequences, before it can be used in practice.

5.2 Open issues

Several issues are, as mentioned, open. In the following we will discuss some of them, it is not intent of this document to cover all open questions.

Relation of cardinalities The definitions of cardinalities so far are not related to each other. That means, the number of instances, and their connection can be freely chosen, even if this is not intended by the designer. This issue will be presumably resolved by introducing a separate *cardinality* element, that can be assigned to several elements. Thus, if the same cardinality is assigned, the instances are restricted, if different cardinalities are used, the designer is free in the instantiation.

Interface changes The declaration of aspect or instance architectures results in the creation or deletion of ports in the architecture. First, new ports are created via ranges during instantiation or weaving. Secondly, ports are created or deleted, if elements are in an *inner* or *identity* relation during weaving.

The creation of ports is not difficult for the structural description of the architecture, but for the conceptual and behavioral description. The concept of component orientation demands for definite interfaces, i.e. ports. The possibility of changing interfaces is therefore not supported very well. This shows up especially in the behavioral description, concepts for describing a variation of ports, even in their existence, are not defined. In order to support these descriptions natively, the descriptions language has to be extended too. A suitable workaround lies in the restriction of the signals of a port: if every port has the same signals, and the ports are identified by parameters, the behavioral description could handle these parameters without a language extension.

Abstract functions vs. open ports During the definition of aspect or instance architectures, often functions are well known to exist, that are needed or that are provided informations to. These functions do not have to be modeled again, the existing functions can be reused. This is the main task for the *identity* relation in combination with abstract functions: identification of existing functions. This results in aspect or instance architectures with abstract functions whose sole purpose is to be identified with existing functions of other architectures.

If one does not like the resulting vastness of the architectures, another solution can be used. In this case, only the needed interface element, port or signal, is

identified with the corresponding target element. The modeling information is the same, it mainly depends on personal modeling favors, which solution is preferred.

Order of weaving In the examples in this document, the order of the weaving, concerning the order of the affected architectures, is irrelevant for the outcome of the weaving process. This is no result of the approach, but originates in the missing behavioral descriptions.

As long as only structure is regarded, the order of the woven architectures remains irrelevant for the outcome. Structural descriptions with the presented elements cannot interfere with each other. The problem arises, if behavioral descriptions are regarded too. In this case, interference can occur, if the related elements are not independent of each other.

The problem of weaving order is well-known in the aspect-oriented community. In this document, we only address the issue and relegate solutions into the existing literature.

Overall behavior One problem, that results from the aspect-oriented approach, and therefore is well-known, is the problem of changing behavior through the weaving process.

By relating elements to each other, the behavior of the resulting architectures is the combined behavior of the elements. In most cases, such a change is wanted, and therefore cannot be avoided. Unfortunately, the resulting behavior can only be observed in the woven architecture. It is not possible to predict the behavior of an unwoven architecture without regarding its relations.

As mentioned before, this is a principal problem of aspect-orientation. The user has to decide, if this becomes a major problem, or if the advantages of the approach overweight this inconvenience.

5.3 Related work

In this subsection we take up the open issues of Section 3.3.3.

In [KTG+06] four requirements for aspect-oriented approaches are given. We do not fulfill requirement 1, because our approach is not UML-based. We chose VEIA

as base models, because they suit the needs of our problem domain. On the other hand, VEIA models can be modeled by UML means as well, therefore, our approach supports UML indirectly. The second requirement demands support for seven main aspect-oriented concepts. Our approach supports so far: aspects, components, point-cuts, advices, static crosscutting, and both relations: aspect-component, and aspect-aspect. So far only dynamic crosscutting is not supported, this will be an issue, when behavior gets into focus. The third requirement demands implementation language independence, until the lowest level of detail is reached. This requirement is fulfilled, because VEIA models are designed to be completely implementation language independent. The fourth requirement of simplicity is fulfilled as well, our approach is practically oriented and implements only features, that are needed in the simplest way possible. Summarized, our approach meets the requirements of [KTG+06].

In [SSK+06] an AOM reference architecture is introduced. It consists of four main building blocks: concern decomposition, language, adaptation subject and adaptation kind. The concern decomposition is reflected in our approach, the principles are realized. The only difference is, that our approach does not differ base and aspect elements. The language is defined as requested, missing the behavior description. The adaptations are realized as well, missing the behavioral elements, too. Therefore, the AOM reference architecture is implemented with our approach, missing only some details, and the behavioral elements.

The systematization of [HOT03] is used to categorize our approach. Concerning element symmetry our approach is symmetric. Every element can be source and target of aspect relations. On the other hand, there are special elements, abstract elements, that do not occur in the woven architecture. The occurrence of such special elements suggests an asymmetric approach. Anyway, the non-distinction between "base" and "aspect" elements is significant for symmetric approaches, therefore, our approach is symmetric. Concerning join-point symmetry, our approach is again symmetric, for only components are join-points so far. This can change, when behavioral join-points are introduced. Concerning relationship symmetry, our approach is asymmetric, for aspects are directly related to components.

The seven issues of [BCG+06] served as reminder, if any points of ADLs or aspect-orientation were not covered with our approach. They are kind of a guideline for defining aspect-oriented ADLs, e.g. Issue 1 "Which ADL elements can have crosscutting concerns?" was answered. The conclusion of [BCG+06] is (in short), that no new architecture elements but connectors and configurations need to be defined in order to enhance an ADL with aspect-oriented elements. As we have shown, we have good reason to introduce further concepts, such as abstract elements, instantiation and so on. This enhances the power of the approach

without making the models too complicated.

[NPMH02] discusses the extensions to an architecture description language, in order to support aspect-oriented modeling. The issues are: definition of join-points, definition of aspects, and definition of connectors for aspects and join-points. Those three issues were regarded in our approach, join-points, aspects, and their connection are defined.

5.4 Pro and Contra

Pro The major advantage of the approach is the possibility to describe overlapping concerns by themselves, and later combine them into the system architecture. Every concern is described independently, resulting in a more clear and subject oriented description. Modeling scopes do not have to be drawn by organizational issues, but can be drawn by functional issues. The interfaces between the concerns are easy to identify.

[ASMT07] indicates, that the benefit of an aspect-oriented approach increases with the size and complexity of the system. Dealing with large, complex systems in the automotive industry, one can assume, that the benefit of this approach will exceed the costs of use.

Furthermore, the separate architectures of the concerns are kept for later modeling, changes, etc. This means, all modeling steps can be made with the original architectures, not the woven model. Thus, traceability of design decisions and modeling elements can be guaranteed with the original design elements, increasing the power to predict impacts of changes.

The approach is an enhancement of the existing VEIA models. The VEIA models implement the basic concepts of component orientation extended by elements for the description of variance and reuse.[7] These basic concepts, except the extensions, are part of every ADL so far. Therefore, the enhancements of this approach can be adopted to other ADLs without fundamental changes. This means, the concepts and ideas of the approach can be used with existing languages.

Moreover, existing modeling tools can be used with the approach, all new elements

[7] The systematization of the models is described in [Man08b].

and relations can be modeled using existing relations, elements, or parameters. The description language changes only in few parts, whereas the result of the weaving process is a common architecture in the original description language. The only missing link is the weaver, that has to be implemented or simulated by hand. Thus, the approach can be integrated into an existing modeling landscape without tool problems.

Contra Like every new modeling technique, this approach has to be learned, understood, and used in the modeling practice. Users of the approach have to be taught the method and encouraged to use it. The approach increases the power of modeling, useful restrictions have to be declared in order to avoid vast modeling.

The approach itself introduces two or more levels of separation of concerns. Every level is modeled independently, increasing the number of models. One has to find a useful balance between the number of models and the diversification of the modeled systems. Too many models lead to complexity, as well as too few models.

As mentioned before, the behavior of the system is not the only sum of its parts, but the combination. This leads to locally correct models, that can behave wrongly after the weaving process. Therefore, the modeling process has to guarantee the local as well as the global correctness of the system architecture.

References

[ABE02] Omar Aldawud, Atef Bader, and Tzilla Elrad. Weaving with statecharts. In *AOSD 2002* [:ao02].

[AEG+06] Omar Aldawud, Tzilla Elrad, Jeff Gray, Jörg Kienzle, and Dominik Stein, editors. *8th International Workshop on Aspect-Oriented Modeling (AOSD-2006)*, March 2006.

[:ao02] *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*, March 2002.

[ASMT07] Francisco Afonso, Carlos Silva, Sergio Montenegro, and Adriano Tavares. Applying aspects to a real-time embedded operating system. In Yvonne Coady, Eric Eide, Marc E. Fiuczynski, Celina Gibbs, Hans-Arno Jacobsen, Julia Lawall, David H. Lorenz, Andreas Polze, Christa Schwanninger, Olaf Spinczyk, Mario Südholt, and Aleksandra Tesanovic, editors, *AOSD 2007*, March 2007.

[Asp] AspectJ-Homepage. http://www.aspectj.org/.

[BCG+06] Thaís Batista, Christina Chavez, Alessandro Garcia, Awais Rashid, Cláudio Sant'Anna, Uirá Kulesza, and Fernando Castor Filho. Reflections on architectural connection: seven issues on aspects and adls. In *Proceedings of the 2006 international workshop on Early aspects at ICSE*, pages 3 – 10, 2006.

[BGL04] Eduardo Barra, Gonzalo Génova, and Juan Llorens. An approach to aspect modeling with UML 2.0. In Omar Aldawud, Grady Booch, Jeff Gray, Jörg Kienzle, Dominik Stein, Mohamed Kandé, Faisal Akkawi, and Tzilla Elrad, editors, *UML 2004*, October 2004.

[BPS06] Peter Braun, Jan Philipps, and Bernhard Schätz. Determining compatibility of embedded software components by communication obligations. In Torsten Klein and Heiko Dörr, editors, *Modellierung 2006*, March 2006.

[CvE07] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. The motorola weavr: Model weaving in a large industrial context. In *AOSD 2007*, March 2007.

[dF07] Dionisio de Niz and Peter H. Feiler. Aspects in the Industry Standard AADL. In Omar Aldawud, Walter Cazzola, Tzilla Elrad, Jeff Gray, Jörg Kienzle, and Dominik Stein, editors, *AOSD 2007*, March 2007.

[GBRS06] Alessandro Garcia, Thais Batista, Awais Rashid, and Claudio Sant'Anna. Driving and managing architectural decisions with aspects. In *ACM SIGSOFT Software Engineering Notes*, volume 31, 2006.

[GCB+06] Alessandro Garcia, Christina Chavez, Thaís Vasconcelos Batista, Cláudio Sant'Anna, Uirá Kulesza, Awais Rashid, and Carlos José Pereira de Lucena. On the modular representation of architectural aspects. In *Proceedings of the European Workshop on Software Architecture*, 2006.

[GEKM07] Martin Große-Rhode, Simon Euringer, Ekkart Kleinod, and Stefan Mann. Grobentwurf des VEIA-Referenzprozesses. ISST-Bericht 80/07, Fraunhofer-Institut für Software- und Systemtechnik, Abt. Verlässliche technische Systeme, Mollstraße 1, 10178 Berlin, Germany, January 2007. Projekt VEIA.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GKM07a] Martin Große-Rhode, Ekkart Kleinod, and Stefan Mann. Entscheidungsgrundlagen für die Entwicklung von Softwareproduktlinien. ISST-Bericht 83/07, Fraunhofer-Institut für Software- und Systemtechnik, Abt. Verlässliche technische Systeme, Mollstraße 1, 10178 Berlin, Germany, October 2007. Projekt VEIA.

[GKM07b] Martin Große-Rhode, Ekkart Kleinod, and Stefan Mann. Fallstudie »Condition-Based Service«: Modelle für die Bewertung von logischen Architekturen und Softwarearchitekturen. ISST-Bericht 84/07, Fraunhofer-Institut für Software- und Systemtechnik, Abt. Verlässliche technische Systeme, Mollstraße 1, 10178 Berlin, Germany, October 2007. Projekt VEIA.

[Gro08] Martin Große-Rhode. Methods for the Development of Architecture Models in the VEIA Reference Process. ISST-Bericht 85/08, Fraunhofer-Institut für Software- und Systemtechnik, Abt. Verlässliche technische Systeme, Mollstraße 1, 10178 Berlin, Germany, May 2008.

[Her02] Stephan Herrmann. Composable designs with UFA. In *AOSD 2002* [:ao02].

[HOT03] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, AOSD 2003, March 2003.

[HTO02] William Harrison, Peri Tarr, and Harold Ossher. A position on considerations in UML design of aspects. In *AOSD 2002* [:ao02].

[IBM] IBM Research. Morphogenic Software. Website: http://researchweb.watson.ibm.com/morphogenic/. last visited: 2008-07-11.

[KL05] Sergei Kojarski and David H. Lorenz. A top-down model of an aop weaving process. In Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, editors, *AOSD 2005*, March 2005.

[Kle06] Ekkart Kleinod. Modellbasierte Systementwicklung in der Automobilindustrie – Das MOSES-Projekt. ISST-Bericht 77/06, Fraunhofer-Institut für Software- und Systemtechnik, Abt. Verlässliche technische Systeme, April 2006.

[KTG+06] Ivan Krechetov, Bedir Tekinerdogan, Alessandro Garcia, Christina Chavez, and Uirá Kulesza. Towards an integrated aspect-oriented modeling approach for software architecture design. In Aldawud et al. [AEG+06].

[Man08a] Stefan Mann. Anwendung von Kohäsions und Kohärenzmetriken auf VEIA-Architekturmodelle zur Bewertung von Produktlinien. ISST-Bericht 86/08, Fraunhofer-Institut für Software- und Systemtechnik, Abt. Verlässliche technische Systeme, Mollstraße 1, 10178 Berlin, Germany, June 2008.

[Man08b] Stefan Mann. *Komposition, Konfiguration und Wiederverwendung von Architekturmodellen eingebetteter Systeme (working title)*. forthcoming, TU Berlin, 2008.

[MR08] Stefan Mann and Georg Rock. Configuration of Product-Line Models in VEIA. Technical Report, Fraunhofer-Institut für Software- und Systemtechnik, Abt. Verlässliche technische Systeme, Mollstraße 1, 10178 Berlin, Germany, June 2008.

[MRW06] Katharina Mehner, Mark-Oliver Reiser, and Matthias Weber. Applying aspect-orientation techniques in automotive software product-line engineering. In *AuRE '06*, 2006.

[MT97] N. Medvidovic and R.N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proc. 6th European Software Engineering Conf. (ESEC/FSE 97)*, pages 60–76. Springer-Verlag, 1997.

[NPM05] Amparo Navasa, Miguel Angel Pérez, and Juan Manuel Murillo. Aspect modelling at architecture design. In *Lecture Notes in Computer Science: Software Architecture*, volume 3527/2005, pages 41–58. Springer, 2005.

[NPMH02] A. Navasa, M.A. Pérez, J.M. Murillo, and J. Hernández. Aspect oriented software architecture: a structural perspective, 2002.

[OT99] Harol Ossher and Peri Tarr. Multi-dimensional separation of concerns in hyperspace. Technical report, IBM T.J. Watson Research Center, 1999.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PFT05] Mónica Pinto, Lidia Fuentes, and José María Troya. A dynamic component and aspect-oriented platform. In *The Computer Journal*, volume 48, pages 401–420, 2005.

[Pre02] Christian Prehofer. Feature interactions in statechart diagrams or graphical composition of components. In Mohamed Kandé, Omar Aldawud, Grady Booch, and Bill Harrison, editors, *UML 2002*, September 2002.

[PRJ+03] Jennifer Pérez, Isidro Ramos, Javier Jaén, Patricio Letelier, and Elena Navarro. Prisma: Towards quality, aspect oriented and dynamic software architectures. In *Third International Conference On Quality Software*, page 59, 2003.

[PSCD06] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A model for developing component-based and aspect-oriented systems. In Welf Löwe and Mario Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2006.

[SGS+05] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 166–175, 2005.

[SSK+06] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, and Manuel Wimmer. Towards a common reference architecture for aspect-oriented modeling. In Aldawud et al. [AEG+06].