

# Towards a Generic Model for AOP (GEMA)

Katharina Mehner<sup>\*†</sup>, Awais Rashid

Computing Department, Lancaster University, Lancaster LA1 4YR, UK

mehner@upb.de, awais@comp.lancs.ac.uk

*Computing Department, Lancaster University Technical Report CSEG/1/03*

## 1 Introduction

### 1.1 Motivation

Aspect-oriented programming (AOP) is a new programming paradigm which aims at improving separation of concerns in programs by providing new kind of modules and new ways of composition. AOP aims at extending existing programming paradigms, most often object-oriented programming languages because of their practical relevance, but it is also conceivable for other programming paradigms such as functional or logic programming. To date, many different AOP approaches have been suggested such as Adaptive Programming [21], AspectJ [2], CompositionFilters [31], Hyper/J [16, 29] and JAC [24]. Although other terms like Advanced Separation of Concerns (ASOC) or Multidimensional Separation of Concerns (MDSOC) are also used to describe these approaches, AOP is the most commonly used term when referring to these approaches and we will use it throughout this report.

While AOP is used to provide a common name for the above-mentioned, non-exhaustive list of approaches, a common agreement on the *essential* characteristics of AOP is still missing. Such an agreement would yield a *definition* of the essential characteristics of AOP which could help to identify whether an existing or new programming approach is AOP or not. For instance, the definition should allow drawing the line between an AOP environment and a meta-object protocol or an OOP environment. A definition also supports establishing the research area of AOP because it can help distinguishing it from closely related but different areas. Having a clear idea what AOP is about is also needed for defining formal semantics (and often formal semantics help with a definition). Since AOP is still an evolving research area giving an ultimate definition is not possible. A working or intermediate definition is desirable which captures the essential characteristics of the state-of-the-art approaches while abstracting from their variability. Of course, each AOP system gives its own definition by providing a concrete approach but this is not what we are aiming for. The only thorough attempt to define AOP independently from a concrete approach is by Filman et al. [10] who define AOP as making quantified assertions about program events oblivious to these assertions. However, this approach does not address modularisation of assertions or structural crosscutting. This will be discussed in more detail in the sequel.

---

\* On the leave of Department of Computer Science, University of Paderborn, D-33095 Paderborn, Germany

† This work was supported by Lancaster University Research Committee Grant FAsOP (Foundations of Aspect-Oriented Programming).

More importantly, a definition is needed as a starting point for developing a *classification scheme*. Any classification scheme for AOP approaches should be able to identify the commonalities of the entities to be classified, i.e., it has to identify the basic concept(s) shared between these entities in order to be able to judge whether they fall into that classification scheme or not. A classification scheme can be even more useful in research and practice than only a definition. Needs for a classification mainly come from two angles: from the users of AOP systems and from the suppliers of AOP systems. For users a common classification is necessary to determine whether a suggested approach is AOP, to see how approaches differ, and to select the best approach for a given program. A detailed classification can help suppliers of tools by identifying commonalities, e.g., in the form of reference architectures, standard components, standardized APIs or data formats, and thereby help in avoiding adhoc-implementations. Ossher et al. undertake efforts in this direction by implementing a concern manipulation layer [15]. Concerning the lack of an essential definition for AOP as a starting point for a classification scheme, we believe that having a classification based on a working definition is better than no classification at all. To the best of our knowledge, such a thorough classification approach is not available at present.

A problem related to the lack of a definition of the essential characteristics of AOP and a classification scheme is the lack of a *common terminology*. At the moment, the terminology differs between the most prominent approaches. This becomes very obvious when approaches are presented and discussed together such as in the recent issue of the CACM on AOP [9]. We illustrate our observations with the example of join point definitions. While G. Kiczales defines join points as points in the execution of a program K. Lieberherr defines them as edges and nodes in a graph which can be a dynamic call graph, a class graph or an object graph thereby generalizing the definition of G. Kiczales. M. Aksit restricts the definition of G. Kiczales to sending and receiving message. H. Ossher's definition covers only static entities such as classes, interfaces, methods and member variables. Albeit all these differences, there is an overlap in the definition of the term join point. We are of the view that these differences can be overcome, probably by merging all the definitions. Other cases however with differing terminology may be more difficult to solve.

To summarize, we have identified three goals for work on foundations of AOP:

- defining the essential characteristics of AOP;
- developing a classification scheme for AOP systems;
- developing a common terminology for AOP.

Note that we are not saying that these are the only topics in foundations of AOP that are worthwhile to explore. Formal foundations of AOP are another important issue. Moreover, aspect-orientation has become a paradigm for all stages of the software development lifecycle such as requirements engineering [12, 26, 27, 28], architecture design [30] and detailed design [4, 5, 13]. However, for the purpose of this discussion, we take AOP literally and focus on programming but not on other phases of software development.

## **1.2 Goals**

The intended scope of this work covers linguistic approaches to AOP and non-linguistic approaches which allow to program in an aspect-oriented programming style, e.g., by offering a restricted meta-object protocol or supporting definition of programming units which act as aspects, for instance in a framework. We will use the term *aspect-oriented programming systems* (AOP systems) for this range of systems and look at systems supporting separation of crosscutting concerns in an object-oriented (OO) decomposition. Also the systems we are going to look at are perceived as general purpose aspect languages, meaning that they are not

domain specific languages supporting, for example, only the domain of persistence or synchronisation.

Our main goal is to work towards a definition of the essential characteristics of AOP. This is not easy because, apart from the terminological differences, approaches differ a lot in the programming mechanisms they provide. Also, a definition needs to be able to cover future AOP approaches. Therefore, a definition has to be independent to a certain extent from existing approaches. To be able to capture existing approaches as well as new ones, we need an *intentional* definition, i.e., a definition that abstracts from concrete approaches but captures the intent of the mechanisms provided. This is opposed to an extensional definition which would be an enumeration based on either the intersection or union of concrete features of different approaches. By providing an intentional definition we also make our definition *generic*. By generic we mean that we want to capture basic yet essential commonalities but can adapt to variability and also optional features. Our approach has been influenced by the ADBMS (Active Database Management Systems) manifesto [7], which distinguishes between essential and optional features and describes them in an intentional style.

The intended classification scheme should also be able to classify future AOP systems. Therefore, the scheme must be formulated in a generic way. The classification scheme should be a refinement of the initial essential definition. Such a classification can be built by comparing a sufficient range of existing approaches in depth to identify a hierarchy of characteristics. Our overall aim is to combine the work on the definition with the work on the classification scheme into one model which we call *Generic Model for AOP* (GEMA). In this report we undertake first steps towards the classification which is to compare two approaches in depth, namely AspectJ [2] and Hyper/J [16].

We will also briefly address our third goal which is the common terminology. We are of the view that AOP systems should be described according to a general and well-established terminology of software composition as found in programming languages and software engineering. We aim to contribute to an understanding of AOP as a modern software composition technique. Therefore, we draw upon well-known concepts such as abstraction, modularisation, etc. These concepts might need to be extended.

The work presented in this report was carried out during a short-term pilot project and is still the subject of ongoing research by the authors.

### **1.3 Overview**

In section 2, we discuss two existing approaches identifying common ideas of AOP. Section 3 provides an initial definition of the essential characteristics of AOP. In section 4, we compare two AOP systems to gain insight for a classification scheme. In section 5, we discuss ideas about software composition and a common terminology. Section 6 highlights some pragmatic issues while section 7 concludes the report.

## **2 Related Work**

AOP systems aim at improving separation of concerns. Some systems focus on better separation at a structural level while others focus on separation of behaviour. We aim at covering this range in the discussion of related work. As mentioned earlier, the only attempt to define AOP, independent from concrete AOP approaches, is by Filman et al. [10]. This work focuses mainly on the separation of behaviour. Therefore, we contrast it with work which takes mainly a structural approach [22]. For each of the two approaches presented, we discuss how well it matches existing AOP techniques before identifying some shortcomings. The comparison of the two approaches is followed by a discussion of the Concern Assembly Tool [15] and the contribution of formal approaches to the definition of AOP.

## 2.1 Quantification and Obliviousness

One of the first and most thorough attempts to identify the common concepts of AOP proposed by Filman et al. identifies *quantification* and *obliviousness* as essential characteristics of AOP. AOP systems allow programming by making quantified programmatic assertions over programs written by programmers oblivious to such assertions. The programmatic assertions contain additional code executed when quantification predicates apply. The quantification is made over the behaviour of programs in terms of an event model. Quantification allows applying the same assertion to more than one place in the code. The one-to-many relationship is also established vice versa. An event can be quantified by more than one assertion. Obliviousness states that the places in the program where the additional code applies are not specifically prepared to receive these enhancements. To allow for obliviousness the event model is not specified by the programmer individually but predefined by the AOP system based on the program entities. Note that obliviousness is not implying that the programmer is not aware that assertions may be added to the program but the programmer cannot specify this apart from using the usual program entities. Filman et al. claim that an AOP approach must also provide the conventional decomposition such as functional or object-oriented decomposition because programs only written as assertions are too difficult to understand. They suggest categorizing AOP systems according to the following three dimensions:

- the kind of quantifications allowed;
- the nature of the actions that can be asserted;
- the mechanisms for composing programs with asserted actions.

We consider this work to be a very important step towards clarifying what AOP is about. Since the assertions contain additional code to be executed when applied, we classify this approach as focussing on providing separation of behaviour. Quantification gives the programmer a means to specify that some assertion does apply to more than one place in the program. Thereby, quantification addresses what is often described as the crosscutting nature of aspects or concerns. Quantification provides a means to express crosscutting of concerns. Obliviousness is often criticized from pragmatic points of view as being too powerful. But the fact is that obliviousness is part of many AOP approaches. For instance, AspectJ advices (together with pointcuts) match the idea of quantification and obliviousness very well. While the definitions given in this work capture many ideas found in AOP approaches, there are also some issues that are not addressed but are essential characteristics:

- Issues of abstraction, structuring and modularisation for the assertion part of the programming model have not been addressed. Concepts addressed by modern programming languages should apply to any part of the AOP system not only to the so-called conventional parts. Such structuring can be found in many AOP approaches and it also makes the assertion part understandable, maintainable, and reusable. Also, it appears that the assertions contain the quantification, i.e., their structure is similar to an Event-Condition-Action rule as found in active database systems [7]. It is not addressed whether the specified events, conditions, and assertions should be decoupled. This is also an issue of modularisation.
- It is the case for many AOP approaches that the assertion part is fundamentally different from the conventional part. For instance, AspectJ uses new kind of modules called aspects and also new kind of functional entities inside these aspects called advice and introduction (weaves). There are also many AOP approaches that do not distinguish between conventional code serving as a base on which to quantify an

assertion. They only separate the composition description part from the rest, e.g., as in Hyper/J.

- It remains unclear if assertions are only applicable to the base code or if they apply also to the code (and the events) from other assertions. For instance, AspectJ allows aspects to generate join points.
- It should be pointed out explicitly that the semantics of the conventional code is no longer the same in the presence of quantified assertions. For instance, a function call in conventional code triggers only the behaviour specified in the function body. In case of object-orientation, the method body is determined dynamically by means of polymorphism. However, in both cases, the caller side is aware of these effects. These paradigms are extended by the assertions. Now, a call can trigger additional assertions.

We finish the discussion by depicting the approach in figure 1 which captures the dimensions of decomposition.

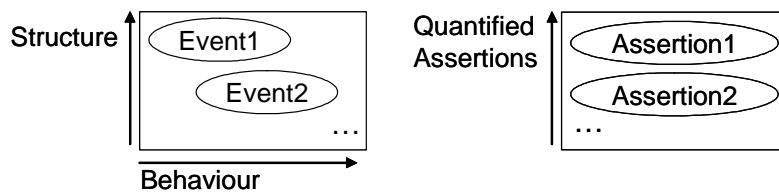


Figure 1: Event-based AOP approach

The figure depicts the conventional code on the left hand side. For Filman et al., conventional code means code structured according to a behaviour-dimension, such as procedures, functions or methods, and to a structure-dimension such as modules or objects. Since this is left open in this approach, we could not depict those entities. The only other assumption made about this code is the fact that it produces events that are linked to the quantified assertions. The right hand side depicts the existence of a set of quantified assertions. The figure cannot depict the semantics of the composition of the quantified assertions with the conventional code. This figure emphasizes the idea that in AOP systems we find code structured according to two different paradigms. Moreover, we observe that the conventional paradigm plays a *dominant* role. That means that the assertions are defined in terms of the conventional code. We will see in the following that this is not necessarily the case. Filman et al. also indicate that a system purely built on these concepts is difficult to program and understand. For some parts of a program what they call a linear and local style, such as found in OOP, is more adequate than the quantification based style.

## 2.2 Concern-Oriented Decomposition

In [22] Nelson et al. describe a model for crosscutting concerns. Figure 2 is taken from their work. Their model has been inspired by the Hyperspaces approach [29]. Although they also address a formalisation based on Labelled Transition Systems, we are only discussing their intentional model for representing crosscutting concerns.

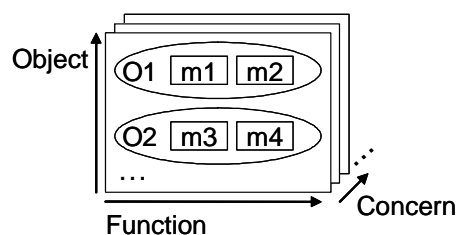


Figure 2: Correspondence-based AOP approach

Their starting point is the object-oriented model which provides decomposition into objects and functional decomposition into methods which belong to objects (methods are depicted as m1 to m4 and objects are depicted as o1 and o2). The object dimension is dominant over the functional decomposition because methods are assigned to objects. Concerns add another dimension of decomposition. Concerns group objects (with their methods). The semantics of composition across the different dimensions is described as follows. Composition can affect structure and/or behaviour. Behavioural composition is mainly accomplished through composition by method calls. Method calls serve two purposes, transfer of control and ferrying of data. Structural composition is mainly accomplished by object-level compositions such as inheritance, association and aggregation. However, object-level composition operators do not only affect structure but also behaviour, e.g., inheritance implies polymorphism. New composition operators structurally and behaviourally compose the concerns based on the assumption that there are entities that appear in more than one concern and thus have to be identified. This is termed *correspondence*. Correspondence restricts the composition between concerns in a way such that it always takes place between entities of same kinds. In addition to the correspondence the behavioural *semantics* (how) and the *binding* (when) of the operators has to be defined. They distinguish two types of correspondence:

- Data correspondence;
- Function correspondence.

*Data correspondence* means that data defined in more than one concern represents the same entity. *Function correspondence* means that different parts of a function are distributed across various concerns, and executing the function should involve all of its parts. In particular, in the case of function correspondence the behavioural semantics have to be addressed, i.e., how all the functions involved will be executed, e.g., in which order or by which selection condition. Binding addresses when and for what duration a correspondence is established.

This model has two main characteristics. Firstly, the composition is described in terms of correspondence which covers structural join points (data correspondence) as well as behavioural join points (functional correspondence). Secondly, the dimensions of concern are treated uniformly, i.e., each concern is structured according to the same dimensions (here objects and functions). This model covers the essence of approaches, such as Hyper/J, which allow encapsulating and composing concerns. We are of the view that it could also be used for Adaptive Programming to describe the correspondence between class graphs. Attaching traversal strategies to class graphs as found in Adaptive Programming can be seen as a high-level correspondence between compositional entities (i.e. the class graphs). Even more generic is the idea of separating collaboration behaviour from the involved entities which also uses object or class graphs as correspondence. Composition by correspondence is therefore independent of a specific layer of abstraction. The two main characteristics might be the cause of problems when trying to apply this model to some AOP approaches:

- The concerns are considered to be uniform according to their internal structure, i.e., objects. This probably needs to be overridden for some approaches such as AspectJ which distinguish between an object or class space and an aspect space which does not use decomposition into objects and methods.
- While correspondence could be used to describe AspectJ's composition by introduction it is very difficult to use with AspectJ composition by advices. Advices are the functional encapsulation within aspects and there is no correspondence with the methods of objects or classes with which they are composed apart from the fact that they encapsulate behaviour.

Some minor problems can be solved by refining or elaborating the approach presented:

- The model suggests that the composition of concerns does not affect the semantics of the other two dimensions which are encapsulated in a concern. However, concern composition extends the semantics of objects and methods because the new mechanisms use them differently than in object-oriented decomposition. The structure of an object may be changed by the composition of concerns, e.g., as a consequence of resolving a correspondence. Similarly, the composition of concerns overrides the execution semantics of method calls such as in AspectJ and Hyper/J. Some approaches explicitly allow fragments of objects and methods in concerns such as in Hyper/J. Fragments do not have semantics in a pure object-oriented approach.
- The model abstracts from the concrete composition operators. Composition is implicit by correspondence. It is not clear whether the model requires correspondence by name or allows explicit approaches as well. Correspondence by name would require a common name space or dictionary across all concerns. Moreover, it could restrict composition of entities from two concerns such that at most one entity from one concern may be composed with one entity in the other concern namely when they correspond by name. A name can only be used once in a concern if there is no qualified hierarchical name space which could solve the problem. Existing approaches such as Hyper/J allow an explicit description of correspondence and thereby allow that an entity from one concern can correspond to several entities in another concern.
- The term data correspondence is problematic. It is not clear what the term data refers to. It could refer to objects or field in objects and therefore needs to be refined.
- Also the axis denoted as object decomposition does not distinguish between object and class level which needs to be refined.

### 2.3 Comparison

We conclude the discussion of the two models for AOP in sections 2.1 and 2.2 respectively with a short comparison. The two models are similar in that they describe means for separation of concerns in addition to procedural, modular and object-oriented decomposition. While Filman et al. focus on behavioural concerns, Nelson et al. address structure and behaviour uniformly. While Filman et al. allow specifying the additional behaviour in a new fashion (or at least impose no restrictions for the assertion part), Nelson et al. require that the same abstractions are used for any concern. Concerns in Nelson's approach are not necessarily oblivious because correspondence can require a common name space. Then the concerns are written with the intended correspondence in mind. Quantification can also be supported in Nelson's approach if the correspondence can be established between several entities. In this case, entities from one concern can apply to several entities in the other concern. Similar to Filman et al., the quantification yields an inverse one-to-many relationship. Because an arbitrary number of concerns can be composed with one concern, an entity from that concern can be merged with many entities from the different concerns. In the following we will refer to this as *multiple compositions* as opposed to quantification. Interestingly, both approaches do not address the potential conflicts which arise from multiple compositions. Since both approaches allow that the same entity be quantified more than once, they need to provide support for conflict resolution which is, e.g., in its simplest form, a default order. Another difference between the two approaches is that correspondence makes an inherent restriction because it allows composition of similar entities only while there is no such restriction between the events quantified and the associated assertions. Therefore, correspondence may be interesting when addressing pragmatic issues for AOP systems (cf. section 6).

## **2.4 Concern Manipulation Primitives**

In an attempt to redesign the implementation of Hyper/J, Ossher et al. describe an interface and a set of concern manipulation primitives implemented in the Concern Assembly Tool [15]. This work identifies primitives of effects of aspects or concerns on other code and thereby reveals the essence of AOP. It can be used to describe existing approaches and also new approaches may be built with it. But this level of abstraction, i.e., the concern manipulation primitives, is too low level to serve as a discriminating classification of AOP. Also, the aim of this work is to provide an implementation backend not a definition for possible front-ends.

## **2.5 Formal Approaches**

Formal approaches are always a means to clarify the meaning of informal descriptions. We feel that the formalisations which exist for AOP are not yet apt for supporting a discriminating classification.

The most comprehensive approach to date is [19]. The authors specify an operational semantics through an interpreter implementation which covers five AOP approaches (CompositionFilters [31], Hyper/J [16], AspectJ [2], Adaptive Programming [21] and a less well-known query language). Regarding its contribution to the classification of AOP and to its definition, this is an extensional characterization by enumerating or merging the operational semantics of approaches. Therefore it does not allow understanding the concepts of AOP, discriminating AOP from other approaches, and deciding whether a new approach is AOP or not. Extensional definitions can not contribute to a general understanding of what is AOP and what is not.

There is also a whole range of approaches trying to formalize the separation of crosscutting behaviour by mapping it to existing formalisations such as Labelled Transition Systems [22] and process algebras [1] or by simulating it in a functional programming language [8]. These approaches abstract from the concrete programming entities and therefore cannot help to intentionally define AOP systems. We are of the view that approaches working on semantics specifically designed for AOP can help to better understand AOP such as the work by Lämmel on method call interception [18] and more recent work on semantic weaving [17].

# **3 Essential and Optional Characteristics of AOP Systems**

In an evolving research area giving an ultimate definition is not possible but a working or intermediate definition is desirable. This is of course developed by looking at a series of prominent approaches but not all details and variability can be considered in such a definition. Here, we try to capture the essential characteristics of the state-of-the-art approaches abstracting from details and variability. As the area evolves our perception of the essential features may change. We also have to clarify why we perceive features of an AOP system as essential. At present, there are no formally established criteria such as Turing-completeness of a programming language. We are of the view that AOP is about expressiveness of programming languages. For a feature to be essential it needs to be useful for improving separation of concerns but also useful for the whole set of other software engineering principles such as readability, reusability, etc. The usefulness and advantages of AOP eventually can only be judged based on case studies and other ways of evaluating AOP systems. Up to now, representative evaluations are still missing. From the experiences so far we can only guess what is really indispensable and therefore essential. Our current perception of essential features is more geared by what is provided by most systems or by what seems to be a systematic extension of existing software composition.

The difficulty with an essential definition is that the ways in which approaches are presented differ namely not only in terminology but also in the viewpoint taken. That is if approaches provide a specific instantiation of a general idea, this general idea is often hidden by their specific instantiation. We have already seen from the two examples discussed in related work that approaches can differ significantly.

### **3.1 Essential characteristics**

Depending on the AOP system the new modularisation concepts have different granularities and the new composition operators support these different granularities and furthermore have different semantics. However, these modules and operators share specific characteristics which distinguish them from existing composition operators for procedural, modular or object-oriented programming.

We are of the view that the composition semantics make the difference, i.e., they define whether an approach is AOP or not. For an AOP approach it is essential that a modularised concern or parts thereof can be composed with more than one other concern or entities in such a concern. This is what has also been termed quantification. We view this characteristic as being independent of the granularity. At some level of granularity there must be a composition operator which supports this requirement. This allows capturing concerns which are generally perceived as crosscutting. The potential separation of crosscutting is what makes an approach AOP. By potentially crosscutting we do not refer to the fact that it might have been scattered in another modularisation but we mean that it can be integrated so that it becomes effective in more than one place in other concerns. Also the style of the quantification can be considered an essential characteristic. Either the individual events or entities quantified are enumerated or there is a declarative, generic or intentional specification of the events or entities to be matched. We are of the view that such a means for specifying the quantification is mandatory. The effect of such a mechanism is that it can also match events or elements which are added to the system later.

We distinguish between *behavioural crosscutting* and *structural crosscutting*. Behavioural crosscutting covers the composition of behaviour such as composing methods, constructors or even behaviour below method level. Structural crosscutting addresses composition of classes by composing their data fields and by introducing new methods into classes. Making this difference is orthogonal to the issue whether it is defined statically, i.e., at compile time or before runtime, or dynamically, i.e., at runtime. In the following we aim at treating structural and behavioural crosscutting equally. We do not differentiate whether a correspondence-based approach, such as the one by Nelson et al., is used or an event-based approach, such as the one by Filman et al., is employed. However, the event-based approach is restricted to be used for behavioural crosscutting.

The idea of crosscutting, as described above, is too complex to hold as an essential characteristic itself. Therefore, we have to look more closely at the means which accomplish it. Crosscutting is addressed by the following essential characteristics of an AOP system. The term component used in the following denotes an encapsulated entity but it does not imply a specific interface or contract as often found with the term component in software engineering.

**Property 1.** An AOP approach provides new components and/or new semantics for existing components to capture and integrate potentially crosscutting concerns.

**Property 2.** These new components have well-defined interfaces where they are composed by the new composition operators. These are generally denoted as join point model. The join point model has to be implicitly defined or predefined for the entire program but not user-defined in general. The join point model covers the interface for structural and behavioural

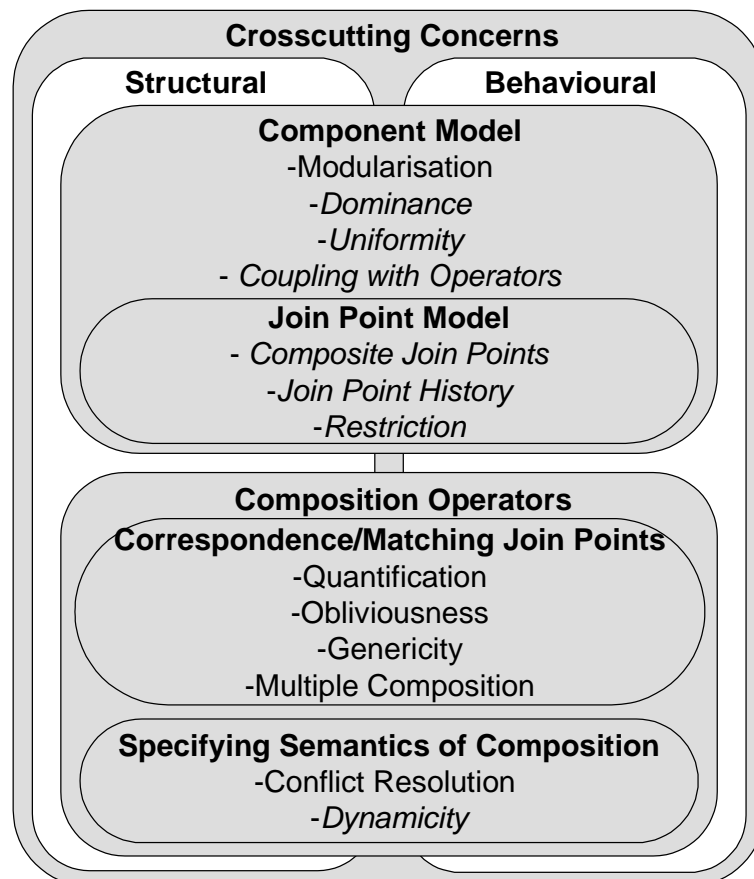
composition. Join points are not the operands but describe at which interface operators can connect the operands.

**Property 3.** In addition to the new components, composition operators are provided which support quantification, genericity, obliviousness, and multiple compositions. They identify the operands of the composition via the join point model and then specify the semantics of the composition and the binding.

**Property 4.** Conflict resolution is mandatory, either predefined or user defined. Conflict resolution plays such an important role because of the possibility of multiple compositions.

**Property 5.** Modularisation concepts for the new components are provided in the case the AOP approach employs new types of components. In the case where mainly old components are used with new semantics these semantics also have to cover existing modularisation concepts, i.e., extend their semantics as well. We are of the view that a plain collection of quantified assertions or composition rules is not compliant with the programmer's expectations of a modern programming environment. One should not go back beyond current principles of software engineering.

As we said, these properties apply to both structural and behavioural crosscutting. The first three properties capture the core idea of AOP while the last two, namely modularisation and conflict resolution make the idea a usable programming approach. Figure 3 shows the general model of AOP and the essential and optional features to be supported by AOP techniques.



**Figure 3. GEMA: A Generic Model for AOP systems (Legend: essential features are in regular text while optional features are in italics)**

We have also identified another potential candidate for an essential characteristic which is *granularity*. By granularity we mean the size of the entities which are composed. Granularity may cover a class graph, a class or method, a fragment i.e. incomplete class or method, or

even an individual data field or a sequence of statements below method level. In the presence of hierarchical structuring of entities, often composition operators are also hierarchical. While the operands are taken from one layer of the hierarchy the operators may recursively operate on lower layers. Thus, the granularity of interest is the set of smallest entities in a hierarchy which are manipulated during composition. For instance, in Hyper/J the composition rules describe composition of concerns which effectively operates on classes and class members. Therefore, the smallest granularity is at the level of methods and data fields. This perception of granularity applies to the structural crosscutting as well as to the behavioural crosscutting. If the granularity is not identified per se as in the correspondence-based approach it is implicit in the event-based approach. Events typically correspond to the units of execution they trigger such as a method call or an exception handler or even an assignment statement. Moreover, the events are often raised before the corresponding behaviour is executed. The corresponding units of execution then become the units which can be manipulated, either augmented or even overridden by the behaviour they are composed with. For instance, in AspectJ, an advice can act upon the execution of a field access and the advice can be triggered before, after, or around the field access. At the moment we have left it as an open issue whether granularity is an essential feature.

### **3.2 Optional features**

Here we list some of the optional characteristics of AOP systems. Some of them are found in existing AOP systems:

- Dominance is an optional feature related to the component model (property 1). It is not essential that components can be distinguished as dominant or non-dominant. This is found e.g. in AspectJ where the object-oriented parts of the program are the starting point for specifying aspects. Sometimes they are also called the base concern. Dominance could imply a default order or selection mechanism during conflict resolution. Dominance may help for humans to understand systems better sometimes. On the other hand, it limits reuse because it may imply a particular use for some of the components.
- Uniformity is also an optional feature related to the component model (property 1). It is not essential that the components are of uniform structure. For instance, Hyper/J treats all concerns uniformly. Uniformity is related to dominance. It is likely that an approach with a dominant concern also tends to be non uniform, i.e. that dominant concern exhibits different concepts than the other concerns.
- Restriction of the predefined join points is an option to extend the join point model. It could forbid join points to be active and thus act against obliviousness but in a well-defined generic way. It is discussed controversially. We do not know of such a system.
- Composite join points are, for example, supported by AspectJ. Join point history is also supported but only to a limited extent.
- Dynamicity can be seen as a feature on its own or as a sub-dimension of composition. For instance, the JAC approach supports dynamicity [24].
- Separate description of composition specification and of the code to be composed. A decision must be made specifically for the quantification predicates or correspondence predicates. They can either be modularised together with the operand they compose or be totally independent.

The optional features are shown in figure 3 alongside the essential ones.

## 4 Towards a Classification Scheme

For the classification, we have focussed on describing the AOP support as is i.e., the mainly linguistic means which are available today. We do not discuss whether there are patterns or workarounds to simulate a mechanism which is not there in the first place. For instance, in AspectJ decoupling of the code to be eventually executed at join points can be decoupled from join points and pointcuts by keeping the aspects minimal and by encapsulating the code in classes. This might be a recommended programming practice but this is not the issue.

For the classification we are not addressing implementation issues such as the time when the composition is carried out (also referred to as weaving time). This is only addressed in so far as it concerns support of specific features. For instance, dynamic aspects are likely to need a runtime support.

In this section we compare two approaches, namely AspectJ and Hyper/J to gain more insight in specific instantiation of some of the essential characteristics discussed in section 3. We mainly focus on behavioural crosscutting though structural crosscutting is also discussed. From the comparison we deduce some possible classification criteria. We also use our knowledge about other approaches, which we can not discuss in depth here, to extend these criteria. Note that this comparison and results deduced from it are by no means complete and several other approaches need to be considered. To start with, we give a very brief overview of AspectJ and Hyper/J.

AspectJ modularises crosscutting code by means of aspects. An aspect is a collection of advices and inter-type declarations (also called introductions). An advice consists of a pointcut designator declaration which is a set of join points, and of the semantics of the advice, i.e., in which order it is executed with respect to other code, and of the body which contains the actual code. In AspectJ, join points are events in the execution of code. An inter-type declaration is a data field or a method to be introduced into a class or a new inheritance relationship. Here, the join points are classes. In addition, an aspect contains fields and methods (similar to classes) which can be used in advice and inter-type declarations. However, in the current version of AspectJ, an aspect cannot be explicitly instantiated though one can specify whether an aspect should be a singleton or a separate instance is created for each join point it affects. AspectJ does not support runtime specification of composition. Figure 4 depicts the general structure of AspectJ but not the details of the composition.

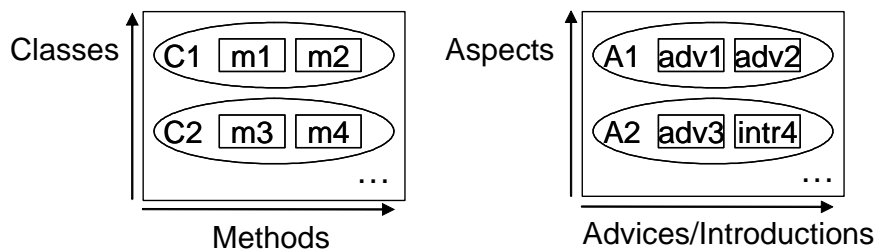


Figure 4. AspectJ

Hyper/J organises code as a set of hyperslices and composition rules (cf. figure 5). Each hyperslice can contain complete classes or their fragments. The class hierarchies are not prepared for composition which is completely specified in the rules. Join points are classes and class members. Roughly, the rules describe which classes correspond and also solve structural conflicts and merge method bodies. Hyper/J defines all correspondences statically.

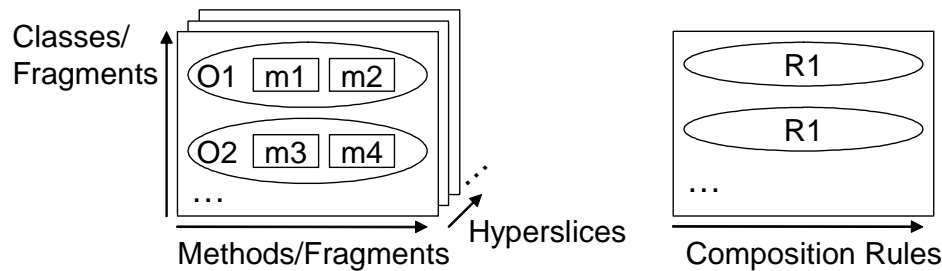


Figure 5. Hyper/J

We will start by comparing the means for behavioural crosscutting. We will look in detail at the essential properties. We will first look at the join point model, then look at the matching part of the composition operators which is responsible for achieving quantification and genericity, then the semantics. We haven't explored further criteria for dominance, obliviousness and uniformity yet nor have we established further criteria of dynamicity.

#### 4.1 Behavioural Join points

In the model of behavioural crosscutting *join points* are actions or events in the execution of a program. Currently, the term program in this definition covers multithreaded but not distributed programs. Distribution is not supported in that one concern can not handle distributed events. Join points can also be generated by aspects when they contain ordinary code. Join points in AspectJ are of the following types:

- method call/execution of public and non public methods;
- object creation, i.e. constructor calls and initialisation including static initialisation;
- read and write access to fields;
- exception handling.

In Hyper/J behavioural crosscutting is considered as part of the entire crosscutting which we will consider in the next section. However we will try to discuss it separately. We find the following join points related to actions in the program execution:

- method execution.

First of all the difference in the two approaches reveals what we have already said. Behavioural crosscutting can be expressed by correspondence as well as by event quantification. AspectJ reveals that there are two different kinds of join points. Structurally defined join points correspond to explicitly encapsulated execution units in the code such as a method. A field access could be seen as just a smaller granularity of encapsulation, i.e., an encapsulation in a statement. But more interestingly, a field access is something semantically or intentionally defined. We know of other approaches which also support property driven join points such as division by zero [11]. But semantic definition could also play a role on a higher level than statement level, e.g., on method level. The distinction between methods, constructors, etc. bears already a semantic differentiation but only one which is syntactically supported. Therefore structural and semantic definitions of join points are a further classification criteria.

##### 4.1.1 Matching Join Points

In AspectJ, the means of identifying or matching join points is called a *pointcut*. A pointcut is a set of join points. The pointcuts are able to match all the possible join points from the above mentioned list. There are different ways of describing those sets.

A pointcut can be a list of *concrete* join points identified by a concrete method name, field name or exception name. A keyword is needed to distinguish special execution semantics.

```
execution(void.Foo.m(int) //execution of the method void.Foo.m(int)
set (Point.x) //when field x of Point is assigned
```

A pointcut can also be described *generically* by providing a kind of regular expressions with the \*-operator for the names of concrete join points.

```
set (!private *.Point*) //when any nonprivate field of Point is assigned
```

There are even more *sophisticated* pointcuts. There is a list of predefined pointcuts which return the set of all joinpoints (of the aforementioned types) which match a selection criterion. This can be all join points in a piece of package, a class or a method.

```
within(package1.package2.*)//all join points in the specified packages
```

This can be all join points in the control flow of an execution.

```
cflow(call(void.Foo.m())//all join points in the execution of the call to Foo()
```

Other matches are possible where a predicate is true or where the arguments of a call or the object calling or called match certain types.

In Hyper/J matching is per se lexical by name correspondence. Concerning behavioural join points it is less powerful because it is not based on an event model but on the functional entities in the source code. A \*-operator exists. To support matching, it allows renaming.

```
Relationship: mergeByName//choose generally strategy
```

```
match class Logging with "*"//match and merge Logging into all classes
```

These are interesting alternatives for a classification scheme. It reveals also that most matching predicates are using lexical matching. A classification could also classify intentional matching predicates. Another category is history of execution.

### 4.1.2 Semantics

During composition of behavioural concerns it is essential to define the order of the composed behaviour. Typical semantics for composing behaviour are

- order
- merge
- override
- select

AspectJ allows three different ways of joining behaviour. Either the new behaviour is executed before, after or around the existing join point behaviour. Around may choose to not even call the existing behaviour. No interleaving can be specified. The statements of the advices are all executed one after the other and also the statements of the method call or the like are executed immediately after another. AspectJ semantics are declared in advices which consists of the corresponding keyword followed by a pointcut as in the following example.

```
before: get (int Point.x { //body contains behaviour }
//runs before field assignment
```

Matching join points also provides information about these join points which go beyond the information which was used to select them. They can be enriched with additional information about the status of a system. It is important that they provide sufficient information. AspectJ allows to access join point information in the additional behaviour by means of `thisJoinPoint`.

This information is accessible from inside the advice body. References, arguments, and signature of advice are accessible.

Hyper/J supports merge, overriding and also a specific way of merging methods using the bracket statement. It can be specified for a method which should be executed before and after another method.

```
bracket Point.*
    from action Application.main
    before Logging.logBefore($OperationName),
    after Logging.logAfter($OperationName);

//makes an entry to the log before and after any method execution of Point called
within application. main
```

In Hyper/J the composition rules would be the place to specify additional conditions. But because of their focus on static integration they are not providing conditions depending on the program state. In the bracket statement a condition can be specified using the from-clause. The from-clause restricts the application of the bracket to only those methods called from within a set of methods specified.

### 4.1.3 Conflict Resolution

Conflicts can arise when multiple concerns or aspects are integrated with each other. Here we will shortly elaborate on the way how AspectJ and Hyper/J deal with it.

AspectJ is not very explicit about conflict resolution. The default semantics is to make no assumptions about a specific order among aspects. They provide one construct to influence order yields a partial ordering.

```
aspect A dominates (B//C) {...} //meaning advices in A have precedence
```

The default in Hyper/J is also to run merged methods in any order. It also provides partial ordering. Order can act on method level but also on higher level thereby providing a default ordering for composition.

```
order action Point.move before action Logging.add; // move executes before add
```

Another interesting approach which discusses conflicts in more depth is JAC [24]. Here we find a list of conflicts. They distinguish the following:

- Checking for compatibility with the application;
- Checking for inter-aspect compatibility and dependence;
- Checking for aspect redundancy;
- Ordering.

We view this as a good starting point for classification criteria.

### 4.1.4 Modularisation Concepts

AspectJ provides inheritance and interfaces for aspects thus taking over standard object-oriented modularisation concepts. The shortcomings of AspectJ are on another area. Because AspectJ ties together pointcuts, advice semantics, and the body reusability is limited. The semantics of the combination with method behaviour are explicitly tied to the advice and its body. Moreover, aspects cannot be used as objects, i.e. their advices are bound to be used as advices. Of course, using object-orientation as base modularisation, advice semantics and advice behaviour can be further decoupled by encapsulating advice behaviour in standard

methods. Despite this shortcoming, the AspectJ approach can improve understanding. Everything about the crosscutting code is specified in one place, i.e., the intention of the code is specified with the code. By intention we mean its intention to be used as a crosscutting concern.

Hyper/J instead uses fragments of well-known modularisation concepts of object orientation. It does not only use fragments but also extends their semantics because they can be composed in more ways. Hyper/J completely replaces the standard object-oriented entities. To be precise an object-oriented program as is can be used in Hyper/J in a standard way but also in a new way, i.e., in a crosscutting way. The disadvantage for understanding is that this not explicit in the code. Hyper/J has a separate composition language. The problem is the structuring of the rules expressed in this language. Specifications become large and are difficult to maintain. There are no ideas yet how to deal with this part of the code.

We have revealed different modularisation concepts and in particular different concepts for coupling of composition semantics with concern semantics.

## 4.2 Structural Join Points

In AspectJ data correspondence is supported by what used to be called introduction and now is called inter-type declaration. Basically, new members (fields and methods) can be introduced to classes. An existing class can be extended with another. An interface can be added to an existing class. Checked exceptions can be converted into unchecked ones.

```
Private int point.x = foo(); //introducing a field with initialisation
```

```
Declare parents = square extends rectangle;
```

```
//legal if rectangle extends original super class of square
```

Because they work below the class level, they don't need to provide sophisticated operations. They only provide additive combination. Their only approach is to identify the class and then to add new elements. However, it can cause conflicts with existing structures, e.g., by name.

The Hyper/J model for data correspondence is much richer than the AspectJ model. Hyper/J uses the term join point model also for establishing data correspondence. Hyper/J allows merging partial class graphs. Changing the inheritance hierarchy explicitly is not possible.

```
relationship: mergeByName
```

```
//overall strategy to merge entities which correspond by name
```

While we can get ideas for the range of structural join points we can only establish that they are structurally defined join points and that their matching is lexical, in a fashion similar to behavioural join points. With Hyper/J we find a set of predefined merge strategies which could also be a criterion for classification.

## 5 Terminology

In this section we describe some initial thoughts on applying software composition principles to AOP.

The history of programming languages is about improving *abstraction* to facilitate the development of large scale complex software systems. Abstraction is the key principle: ignoring what is irrelevant in order to focus on the essence of a problem. Thereby abstraction is relative to the perspective taken. It helps to master complexity by reducing complexity. Abstraction is often used in a layered way and this is why we often speak of raising the level of abstraction. Abstraction lies at the heart of many ideas and concepts for software development or software engineering. Some examples of abstractions are expressions, data abstractions, procedures, objects, functions, abstract data types, modules and interfaces,

inheritance, and components. One can distinguish structural and behavioural abstractions. For example, inheritance affects structure and behaviour. The polymorphism introduced with inheritance is about abstraction, i.e., decoupling the message from its body.

One important application of abstraction is the *separation of concern* principle: dealing with one important issue at a time [6]. It can be seen as a principle which has to find good or useful abstractions among the many possible ones. A closely related important principle is *decomposition in manageable parts* [23]. Decomposition can be of different kind and then we speak of different dimensions of decomposition. Abstraction is usually the driving force behind decomposition. Abstraction comes to play during composition because it allows for composition of the parts by looking only at the important parts of the parts. Each composition provides a specific abstraction, e.g., association, aggregation, inheritance. While aggregation and association rely on the same interface they are semantically different and make different assumptions. Inheritance uses an extended interface. Here we look at some typical principles of components and composition which are of dual nature.

Typical dimensions of components are:

- modularisation;
- encapsulation;
- information hiding.

Typical dimensions of composition are:

- coupling;
- cohesion;
- time and duration.

Composition can achieve what we generally perceive as structuring. This is usually applied to static composition only. Some structural composition operators can be applied recursively which yields layers such as inheritance.

The dimensions of structuring are:

- hierarchy;
- grouping;
- network;
- sequence;
- nesting.

Often, AOP is viewed as a third dimension of decomposition in addition to object-oriented decomposition (inheritance hierarchies) and functional decomposition (methods and modules). AOP provides various forms of abstraction. Obliviousness is abstraction. Objects abstract from aspects, i.e., they don't know about them. Aspects abstract from object implementation. Quantification is also abstraction. Aspects thereby abstract from the objects they apply to. The aim is to integrate the separation of crosscutting achieved by AOP systems into the dimensions of components, composition, and structuring.

## 6 Pragmatic Issues

In this section we discuss some pragmatic issues of the characteristics which we have dealt with so far. While this is not part of our generic AOP model, this discussion might bring to front proposals for new features.

An issue often addressed is lexical matching of join points. This has been considered to be less powerful when system changes are made. Therefore future work should explore more powerful intentional or semantic join points such as in the work by K. Gybbels [14]. Also, intentional specification may improve understanding by the programmer.

This leads us to another idea. Aspects or concerns are probably better understood when separated. But they are also a burden to understanding an entire system because of the obliviousness and the quantification characteristics. They can be effective everywhere in the system. Although this power is desired and a key characteristic, in order to better manage complexity one might be interested in limiting the scope of aspects or to limit the join point model. This is a standard principle to encapsulate subsystems and provide information hiding. Such features are discussed controversially because while they might help building systems from scratch they hamper use of AOP for unanticipated change. Our answer to this is that the software industry demands configurable programming systems. This trend is already demonstrated in the context of AOP by the advent of hybrid approaches [3, 25].

Self-application and recursion is an issue from a software engineering point of view. Aspects may introduce new join points and end up in recursion. It is probably one common error people will have to face with. The question is whether recursion is needed for aspects for expressiveness or completeness or if it is only harmful.

## 7 Conclusion and Outlook

We have identified needs for work on foundations of AOP. The topics we have motivated are definition, classification and terminology. We have carried out initial work which can be guidance to these topics. A common understanding of AOP is essential not only to promote uniformity of concepts but also to compare and contrast AOP approaches. Such a common model can also serve to improve tool support. For instance, in [20] we have proposed the use of common models for debugging and tracing of AOP systems.

Our work in the future will focus on the refinement of the generic AOP model discussed in this paper and its use to carry out an in-depth comparison of various AOP approaches available.

**Acknowledgement:** The authors wish to thank members of the Aspect-Oriented Software Engineering Group at Lancaster especially Ruzanna Chitchyan and Mattia Monga for many fruitful discussions on the topic.

## 8 References

- [1] J. H. Andrews, "Process-Algebraic Foundations of Aspect-Oriented Programming", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 187-209.
- [2] AspectJ Team, "AspectJ Project", <http://www.eclipse.org/aspectj/>, 2003.
- [3] R. Chitchyan, I. Sommerville, and A. Rashid, "An Analysis of Design Approaches for Crosscutting Concerns", Workshop on Aspect-Oriented Design (held in conjunction with the 1st Aspect Oriented Software Development Conference AOSD 2002), 2002.
- [4] S. Clarke and R. J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects", International Conference on Software Engineering (ICSE), 2001.

- [5] S. Clarke and R. J. Walker, "Towards a Standard Design Language for AOSD", 1st International Conference on Aspect-Oriented Software Development, 2002, ACM, pp. 113 - 119.
- [6] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall, 1976.
- [7] K. R. Dittrich, S. Gatzju, and A. Geppert, "The Active Database Management System Manifesto: A Rulebase of ADBMS Features", 2nd Workshop on Rules in Databases, 1995, Springer-Verlag, Lecture Notes in Computer Science, 985, pp. 3-20.
- [8] R. Douence, O. Motelet, and M. Südholt, "A Formal Definition of Crosscuts", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 170-186.
- [9] T. Elrad, R. Filman, and A. Bader (eds.), "Theme Section on Aspect-Oriented Programming", *Communications of ACM*, Vol. 44, No. 10, 2001.
- [10] R. Filman and D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", OOPSLA WS on Advanced Separation of Concerns, 2000.
- [11] P. Fradet and M. Südholt, "An Aspect Language for Robust Programming," Workshop on Aspect Oriented Programming (held in conjunction with European Conference on Object-Oriented Programming), 1999.
- [12] J. Grundy, "Aspect-Oriented Requirements Engineering for Component-based Software Systems", 4th IEEE International Symposium on RE, 1999, IEEE Computer Society Press, pp. 84-91.
- [13] J. Grundy, "Multi-perspective specification, design and implementation of software components using aspects", *International Journal of Software Engineering and Knowledge Engineering*, Vol. 20, No. 6, 2000.
- [14] K. Gybels and J. Brichau, "Arranging Language Features for More Robust Pattern-based Crosscuts", 2nd International Conference on Aspect-Oriented Software Development, 2003, ACM, pp. 60-69.
- [15] W. Harrison, V. Kruskal, H. Ossher, P. Tarr, and F. Tip, "Common Low-level Support for Composition and Weaving", Workshop on Tools for Aspect-Oriented Software Development (held in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA 2002), 2002.
- [16] IBM Research, "Hyperspaces", <http://www.research.ibm.com/hyperspace/>, 2003.
- [17] R. Lämmel, "Continuation Semantics based Weaving", Belgian and Dutch AOP Workshop, Twente, The Netherlands, 2003.
- [18] R. Lämmel, "A Semantical Approach to Method-Call Interception", 1st International Conference on Aspect-Oriented Software Development, 2002, ACM, 41-55.
- [19] H. Masuhara and G. Kiczales, "Modelling Crosscutting in Aspect-Oriented Mechanisms", European Conference on Object-Oriented Programming (to appear), 2003.
- [20] K. Mehner and A. Rashid, "Towards a Common Interface for Runtime Inspection in AOP Environments", Workshop on Tools for Aspect-Oriented Software Development (held in conjunction with ACM Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA 2002), 2002.

- [21] M. Mezini and K. J. Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development", OOPSLA, 1998, ACM, SIGPLAN Notices, 33(10), pp. 97-116.
- [22] T. Nelson, D. Cowan, and P. Alencar, "Supporting Formal Verification of Crosscutting Concerns", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 153-169.
- [23] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, Vol. 15, No. 2, 1972.
- [24] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 1-25.
- [25] A. Rashid, "A Hybrid Approach to Separation of Concerns: The Story of SADES", 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), 2001, Springer-Verlag, Lecture Notes in Computer Science, 2192, pp. 231-249.
- [26] A. Rashid, A. Moreira, and J. Araujo, "Modularisation and Composition of Aspectual Requirements", 2nd International Conference on Aspect-Oriented Software Development, 2003, ACM, pp. 11-20.
- [27] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo, "Early Aspects: A Model for Aspect-Oriented Requirements Engineering", IEEE Joint International Conference on Requirements Engineering, 2002, IEEE Computer Society Press, pp. 199-202.
- [28] S. M. Sutton and I. Rouvellou, "Modeling of Software Concerns in Cosmos", International Conference on Aspect-Oriented Software Development, 2002, ACM, pp. 127-133.
- [29] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", International Conference on Software Engineering (ICSE), 1999, ACM, pp. 107-119.
- [30] B. Tekinerdogan, "ASAAM: Aspectual Software Architecture Analysis Method", Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (held in conjunction with AOSD 2003), 2003.
- [31] University of Twente, The Netherlands, "TRESE Home Page", <http://trese.cs.utwente.nl/>, 2003.