A Theory of Aspects for Aspect-Oriented Software Development

Christina von Flach G. Chavez[1,2] *Carlos J. P. de Lucena*[2]

[1]UFBA, Computer Science Department
Av. Adhemar de Barros, s/n – 40170-110, Salvador, BA, Brazil
flach@ufba.br

[2]PUC-Rio, Computer Science Department, SoC+Agents Group
R. Marquês de São Vicente, 225 – 22453-900, Rio de Janeiro, RJ, Brazil
lucena@inf.puc-rio.br

Abstract

Aspect-Oriented Software Development (AOSD) is an emerging area with the goal of promoting advanced separation of concerns throughout the software development lifecycle. However, since there are many different approaches to Aspect-Oriented Programming (AOP), it is very difficult to identify the essential concepts and properties for supporting the design of aspect-oriented languages and tools, and effectively promote advances in the field of AOSD. In this context, this paper presents a *theory of aspects* – a conceptual framework for AOP that provides consistent terminology and basic semantics for thinking about a problem in terms of the core concepts and properties that characterize the aspect-oriented style as an emerging paradigm to software development. This theory has been used for evaluating existing aspect-oriented languages and tools and, in special, it has been used to characterize `aSide`, an aspect-oriented modeling language.

Keywords: Aspect-oriented software development, aspect-oriented language, aspect model, conceptual framework, separation of concerns.

1 Introduction

The evolution of a new software engineering paradigm often progresses from programming towards design and analysis, to provide a complete path across the software development lifecycle. *Aspect-Oriented Programming* (AOP) is reaching maturity after almost a decade of research [14, 20, 24, 16, 15] and a growing number of applications, tools and users. In this context, Aspect-Oriented Software Development (AOSD) naturally emerges to foster the goal of promoting advanced separation of concerns from implementation level to other stages of the software development process, including requirements specification, analysis and design.

AOP considers acknowledged improvements to separation of concerns provided by previous technologies (mainly object-oriented programming, but not constrained to it), while supporting new mechanisms to deal with *crosscutting concerns*, that is, special concerns that are not properly modularized by these technologies. AOP introduces *aspects* as a new modularization mechanism for separating crosscutting concerns and provides a new composition mechanism for *weaving* aspects back into components at well-defined *join points*.

The use of separate linguistic mechanisms to modularize and compose crosscutting concerns has been reified by some other approaches, not just AOP. Related work includes *Adaptive Programming* (AP) [19], *Composition Filters* (CF) [4], *Subject-Oriented Programming* (SOP) [12] and *Multi-Dimensional Separation of Concerns* (MDSoC) [27]. AOP and these approaches belong to a research area known as *Advanced Separation of Concerns* (ASoC). However, AOP and related ASoC approaches propose distinct and varying sets of abstractions and composition mechanisms, with specific terminology, properties and language constructs.

Recently, AOP has been regarded as a possible convergence of these independent ASoC research paths [9], but the definition of an unifying conceptual framework for AOP that can also be used across other approaches is still missing. The adoption of an unifying conceptual framework for AOP is an important step for characterizing the design space of aspect-oriented languages and providing support for aspect-oriented software development.

This paper presents a disciplined, yet still informal, *theory of aspects* – a conceptual framework for aspect-oriented programming that provides consistent terminology and basic semantics for thinking about a problem in terms of the *concepts* and *properties* that characterize the AOP style as an emerging paradigm to software development. These concepts and properties have already been described informally by their authors [16, 11, 10]. Our goal is not to present a survey of existing concepts used by different approaches for ASoC but, instead, to present the defiinition of an ontology that, according to our point of view, subsumes the essential concepts and properties for supporting the design of aspect-oriented languages and the development of aspect-oriented software systems. In such informal setting, our theory of aspects consists of a description in natural language of categories of core concepts, properties and rules that these concepts must satisfy, as well as a set of entity-relationship conceptual models [7] – following the approach proposed by the Theory-Model paradigm [29]. The theory of aspects can be used for evaluating existing *candidate* aspect-oriented languages and tools as well as for driving the design of new aspect-oriented languages.

The rest of the paper is organized as follows. In Section 2 we present our theory of aspects, that for historical reasons, we call the *aspect model*. In Section 3, we use the *aspect model* to present a definition for aspect-oriented languages (Section 3.1) and to characterize some representative approaches to advanced separation of concerns (Section 3.2). We also provide a brief description of how the *aspect model* can be used to support the design of an aspect-oriented language. In the last Section, we present our conclusions, related and future work.

2 The Aspect Model

Aspects, *components*, *join points*, *crosscutting* and *weaving* are concepts that have been introduced by Kiczales et al. in their seminal paper *Aspect-Oriented Programming* [16] and collectively constitute the heart of the aspect-oriented paradigm. Additionally, two properties, *quantification* and *obliviousness*, have been proposed as necessary properties for AOP [11]. Following the idea of adopting AOP as a possible convergence of independent ASoC research paths [9], we consider these concepts and properties, as well as the clear separation between aspects and components [17] – which we call the *aspect-base dichotomy* – as fundamental elements of the aspect-oriented paradigm. We adopt them as the core conceptual framework that characterizes everything that is *aspect-oriented*. We have organized these elements into four interrelated conceptual models: (i) the *component model*, (ii) the *join point model*, (iii) the *weaving model*, and (iv) the *core model*. Following this pattern, we call the resulting composite model the *aspect model* (see Figure 1). As a first approach, we define that an *aspect-oriented language* is a language that supports the *aspect model.*

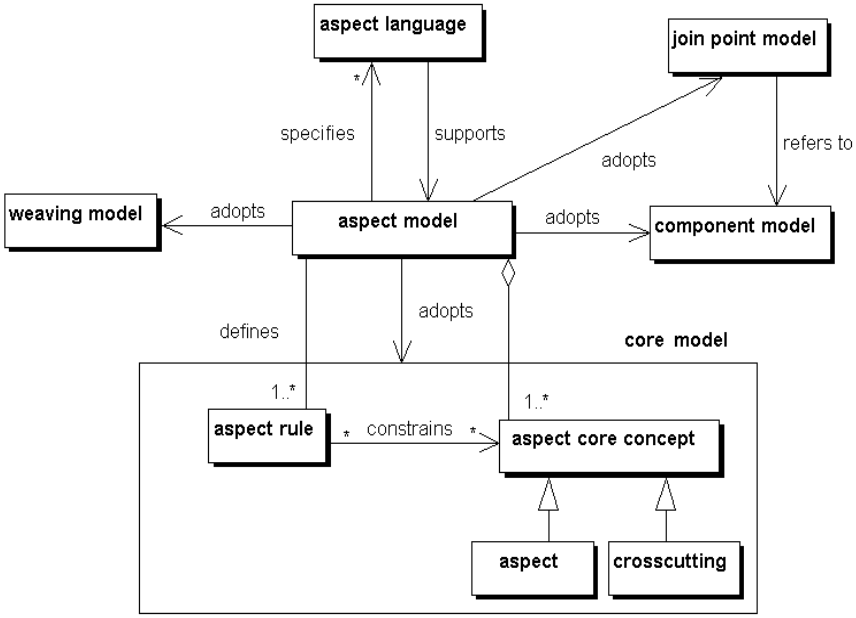

Figure 1: The *aspect model*

In the following Sections, we describe the conceptual models (Sections 2.1 to 2.4) and discuss AOP properties (Section 2.5). We use entity-relationship diagrams to illustrate each conceptual model in terms of entity sets and relations over these sets. For each core concept, we present existing definition(s) as a starting point to provide our own definition and terminology, followed by a succinct discussion and some examples.

2.1 The Component Model

The *component model* represents a conceptual framework used for thinking about a problem and decomposing it in terms of a certain kind of component. This framework consists of categories of core concepts (*components* and *composition mechanisms*), rules that constrain elements of those categories and a set of general principles.

Definition. A *component* is an unit of the system's functional decomposition, a property or concern that can be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API) [16]. We use the term *component* to denote a nameable entity that modularizes a *functional part* of a software system, more or less independent from other parts, and that can be naturally composed with other components using the provided composition mechanisms.

Discussion. The component model is a fundamental part of the *aspect model.* The state of being a crosscutting concern is relative to a particular kind of decomposition. The component model defines the primary kind of decomposition and a component language is used to express non-crosscutting concerns effectively. The component model also constrains the design space of the join point model (see Section 2.2).

A component model may be supported by one or more *component languages*. Figure 2 presents a data model for the component model.

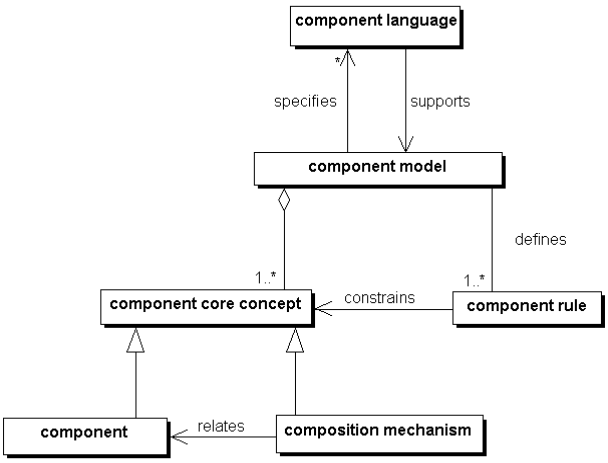

Figure 2: The component model

Example. Components can be reified by functions, procedures or classes; common composition mechanisms are function calls, procedure calls, and method invocations. The *object model*, i.e., the conceptual framework for all things object-oriented [5], is a component model, where the main concepts are objects, classes and inheritance [32]. A typical rule[1] is that every object is an instance of some class. General principles are abstraction, encapsulation, modularity and hierarchy [5]. Java and C++ are object-oriented programming languages since their conceptual framework is the object model, that is, both support the object model. Nevertheless, Java supports single inheritance while C++ supports multiple inheritance, among other differences.

2.2 The Join Point Model

The *join point model* represents a conceptual framework used for describing the kinds of join points of interest and the associated restrictions for their use. The join point model is highly dependent on the adopted component model. Figure 3 presents a data model for the join point model.

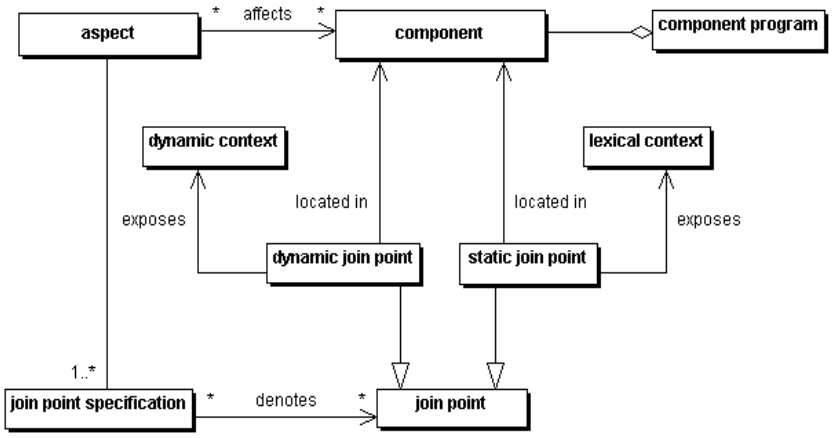

Figure 3: The join point model

[1] for class-based models

Definition. *Join points* are elements of the component language semantics that aspects coordinate with [16]. We use the term *join point* to denote an element related to the structure or the execution of a component program that is referenced and possibly affected by an aspect. A *static join point* is a location in the structure of a component whereas a *dynamic join point* is a location in the execution of a component program.

Discussion. The concept of *join point* refers to some location related to a component, under the aspect's perspective. Components "own" the actual locations, while aspects describe them as join points of interest and use them to affect components with their crosscutting functionality. Join points may expose additional information related to the context where they show up – we call it the *crosscutting context*. The nature of the available context information depends on the kind of join point: it may be *static* (available from program text) or *dynamic* (available from program execution). The aspect weaver combines aspects and components at the join points specified by the aspect (see Section 2.4).

There are many locations in a component or component program execution that can be used as join points, but in practice, only a subset is regarded as useful [26]. Aspect-oriented languages must define their set of join points taking into account their corresponding component language. The selected subset will influence the range of quantification allowed in those aspect-oriented languages (see Section 2.5).

Example. If we consider the object model as a component model, some possible dynamic join points are method calls, method executions, instantiations, constructor executions, field references and handler executions; some static join points of interest are classes, interfaces, methods and attributes. For an aspect-oriented language that adopts other component model, the join point model will certainly be different.

2.3 The Core Model

The goal of AOP is to support the programmer in cleanly separating *components* and *aspects* from each other, by providing *mechanisms* that make it possible to (i) *abstract* and (ii) *compose* them to produce the overall system [16].

The *core model* represents a conceptual framework used for describing *aspects* as an abstraction mechanism, and *crosscutting* as a composition mechanism.

2.3.1 Aspects

Definition. *Aspects* are defined as *system properties that cross-cut components*, and, more specifically, as *properties that affect the performance or semantics of components in systemic ways* [16]. We also use the term *aspect* to denote a first-class, nameable entity that provides modular representation for a *crosscutting concern*.

Discussion. The modularization of a crosscutting concern involves the provision of an abstraction mechanism – the aspect – that localizes both:

- the specification of a particular set of join points, and
- enhancements[2] to be combined at the specified join points.

[2]The term *enhancement* is used here to denote any modification to components, not necessarily in a monotonic way: aspects can also delete behavior.

To establish a language-independent terminology, we use the term *crosscutting interface* to denote the set of join points specified inside the aspect, and the term *crosscutting feature* to denote any structural or behavioral enhancement specified inside the aspect to affect one or more components at the specified join points (see Figure 4).

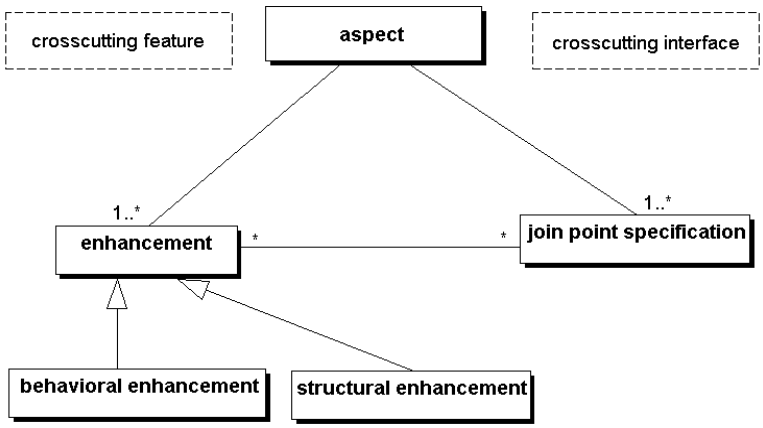

Figure 4: Aspect

Crosscutting Interfaces. *Crosscutting interfaces* comprise *join point specifications*, i.e., they describe the kinds of join points of interest for the aspect, constrained by the adopted join point model.

Crosscutting Features. *Crosscutting features* are attributes and operations that describe enhancements to the structure and behavior of components. These enhancements may *add* new structure and behavior to one or more components, *refine* or even *redefine* existing behavior.

Whenever necessary, we use the term *structural crosscutting feature* to denote structural enhancements and the term *behavioral crosscutting feature* to denote behavioral enhancements. We further distinguish between *static crosscutting features* and *dynamic crosscutting features*, that is, crosscutting features that use static and dynamic joint points, respectively.

	Static Crosscutting	Dynamic Crosscutting
Structural	attributes	
Behavioral	operations	operations

Table 1: Crosscutting Features

Example. The `Logging` aspect is not only the code that actually logs some particular data. The crosscutting concern – and therefore, the aspect – is that *a particular set of points should log some particular data*. The AspectJ language [15] supports this view: aspects are defined as first-class implementation elements that comprise *pointcut declarations* to express join point specifications, and *advice*/*introduction declarations* to express crosscutting features.

2.3.2 Crosscutting

Definition. *Crosscutting* is defined as a phenomena that is observed whenever two properties being programmed must compose differently and yet be coordinated [16]. We use the term *crosscutting* to denote the composition mechanism used to compose aspects and components.

Aspects may crosscut one or more components, possibly affecting their structure and behavior. Furthermore, we broaden the usage of the term to also denote a *relationship* from an aspect to one or more components.

Discussion. From the first definition given above [16], it follows that aspects can crosscut other aspects. Nevertheless, in this core theory we restrict ourselves to crosscutting as a composition mechanism among aspects and components. Furthermore, we distinguish the following issues when regarding crosscutting as a composition mechanism.

Direction of Crosscutting. The direction of crosscutting is always from aspects to components. The term *reverse inheritance* has been used elsewhere to denote the composition mechanism that relates aspects and classes, since the direction of composition is the opposite of conventional inheritance. Although the object model is often the choice for the component model, we prefer to use the term *crosscutting* instead of *reverse inheritance* to preserve the independence of the *aspect model* relative to its sub-models.

Dimensions of Crosscutting. Crosscutting can be applied *homogeneously*, by providing exactly the same set of enhancements to one or more components, or *heterogeneously*, where subsets of the enhancements are applied simultaneously to different kinds of components. We call the first one, *vertical crosscutting* and the latter, *horizontal crosscutting*.

Cardinality of Crosscutting. Aspects may crosscut one or more components, simultaneously. Components may be crosscut by one or more aspects, simultaneously.

The Nature of Crosscutting. We call *static crosscutting* the kind of crosscutting that uses static join points. We call *dynamic crosscutting* the kind of crosscutting that uses dynamic joint points.

Example. A characteristic feature of object-oriented programming is *inheritance* [30]. Among other uses, inheritance is regarded as a mechanism for structural composition that allows new class definitions to be based on existing ones. A new class inherits the existing properties from its parents, and may add new properties, refine or redefine its inherited properties.

A characteristic feature of aspect-oriented programming is *crosscutting*, a composition mechanism that allows aspects to be combined to existing components at well-defined join points. An aspect crosscuts one or more components, and may add new properties, refine or redefine existing properties.

2.4 The Weaving Model

The *weaving model* represents a conceptual framework used for describing the kinds of weaving mechanisms.

Definition. *Weaving* is the process of composing aspects and components related by crosscutting at the specified join points. The term *aspect weaver* designates the tool that composes aspects and components [16].

Discussion. The weaving model is part of our conceptual framework since it can be useful to render evident language characteristics and/or properties, as for example, it is often useful to know that some component language is required to be interpreted or if it can also be compiled. In the following paragraphs, we identify and present some relevant facets of weaving.

Weaver Inputs. An aspect weaver can work on source code, byte code, or object code.

Weaver Outputs. An aspect weaver can provide *in-place modification* or *client migration* [28]. *In-place modification* is destructive, that is, the original component code is no longer available after weaving. *Client migration* means that both the original component and the woven versions are available. Instead of changing permanently the original component, new clients are created that refer to a new woven component.

Weaving Time. Weaving can be *static* (compile or load time) or *dynamic* (run-time). *Static weaving* is a weaving technology in which the component program and the aspect program are merged into a new version of the sources, just before or during compilation. *Load-time weaving* is a special kind of static weaving technology that does not require the source code. The aspect program can be combined to the component program in binary format using *code instrumentation*. *Dynamic weaving* is a weaving technology that allows aspects to be weaved and unweaved during execution.

Example. Most weaving tools for AOP are based on static weaving, either on the source or directly on the object code. The AspectJ compiler (ajc) is an aspect weaver. In its previous incarnations, it worked by preprocessing a set of .java source files that contained aspect descriptions and generating woven .java source files or Java .class files. Now, AspectJ also supports *bytecode weaving*, by allowing as input Java .class files.

AspectJ performs in-place modification, which invasively changes the original components, while Hyper/J, a tool that supports multi-dimensional separation of concerns for Java [13], performs client migration.

2.5 Properties

Aspect-oriented programming encompasses some well-known principles and new properties. The two main principles of AOP are *separation of concerns* and *modularity*. Separation of concerns is reified through the clear separation between aspects and components. Modularity is reified through a new kind of modular unit for crosscutting concerns, the aspect. We discuss both of them below under the label *aspect-base dichotomy*.

Additionally, two properties, *quantification* and *obliviousness* were proposed as the core characteristic of AOP, and have been used to define whether a language is aspect-oriented or not [10].

2.5.1 Aspect-base dichotomy

The *aspect-base dichotomy* stands for the adoption of a clear distinction between components (or *base components*) and aspects. When the aspect-base dichotomy holds, taking into account the principles of separation of concerns and modularity, it follows that:

- Systems are decomposed into components and aspects
- Aspects modularize crosscutting concerns

- Components modularize non-crosscutting concerns
- Aspects must be explicitly represented apart from components and other aspects.

In our conceptual framework, the aspect-base dichotomy is considered an essential characteristic of AOP.

2.5.2 Obliviousness

Obliviousness is the act or effect of being oblivious, in the sense of being forgetful or unaware. In the context of AOP, obliviousness is the idea that components do not have to be specifically prepared to receive the enhancements provided by aspects [11]. When the obliviousness property holds, it follows that:

- Components are not aware of the aspects that will eventually crosscut them
- Programmers do not have to spend additional effort while implementing the component's functionality to make AOP work and to realize its benefits [11].

In our conceptual framework, we adopt obliviousness as an essential property of AOP, to be used as an informal measure that indicates the usefulness of aspect-oriented systems.

Discussion. Obliviousness is considered the feature that makes AOP special [10]. The use of aspects to modularize crosscutting concerns supported by the obliviousness property enhances better separation of concerns in software development and simplifies the analysis, design and implementation of components.

Better AOP systems are supposed to be more oblivious [11]; however, complete obliviousness is a hard goal to be achieved. In this context, *intimacy* is defined as the additional effort required to prepare the components for aspects [9].

2.5.3 Quantification

Quantification is defined as the ability to write unitary and separated statements that enhance many non-local places in a programming system. Quantification provides the capability for stating things like: "in programs P, whenever condition C arises, perform action A" [10].

When the quantification property holds, it follows that:

- Aspects may crosscut an arbitrary number of components simultaneously.

In our conceptual framework, we adopt quantification as an essential property of AOP.

Discussion. The ability to quantify in AOP systems is related to the support for declarative reasoning about several kinds of elements pertaining to a component program, for example, fields, field sets, methods, method calls, etc. This support requires some sort of quantification language that allows the expression of quantified statements. These statements may contain one or more *quantification variables* to be bound to values in some universe of discourse, and an *open statement* that may be evaluated or applied in this context.

In our framework, we define as the universe of discourse elements that belong to the structure of a program (static elements) or to the execution of a program (dynamic elements); therefore, quantification variables can be bound to static elements (*static quantification*) or dynamic elements (*dynamic quantification*). Open statements correspond to behavioral crosscutting features (see Section 2.3.1).

Example. In static quantification, quantification variables are bound to elements such as classes, fields, methods, etc. In dynamic quantification, quantification variables are bound to elements such as method calls, object creation, etc.

AspectJ supports dynamic quantification through *pointcut designators* and quantification variables match points in program execution.

3 Using the Aspect Model

The main benefit of having an conceptual framework such as the *aspect model* is to provide support for assessing existing approaches as well as for developing new methods, languages and tools based on unified terminology, concepts and properties defined in the framework. Table 2 summarizes the concepts and properties that comprise the *aspect model.*

Component model	component, composition mechanism, component rule, component language
Join point model	join point, static join point, dynamic join point, static context, dynamic context
Weaving Model	aspect weaver, weaving, static weaving, dynamic weaving
Core model	*aspect*, crosscutting interface, crosscutting feature, structural crosscutting feature, behavioral crosscutting feature, *crosscutting*, static crosscutting, dynamic crosscutting, horizontal crosscutting, vertical crosscutting
Properties	aspect-base dichotomy, quantification, obliviousness

Table 2: Concepts and properties defined in the *aspect model*

In this Section, the *aspect model* is used to provide a definition for *aspect-oriented languages* (Section 3.1) and to characterize four representative ASoC approaches (Section 3.2). Moreover, we illustrate how the *aspect model* has been used to support the design of `aSide`, an aspect-oriented modeling language (Section 3.3).

3.1 Aspect-Oriented Languages

Definition. An *aspect-oriented language* (AOL) is a language that supports the *aspect model* and therefore satisfies the following requirements:

- it adopts a component model
- it adopts a join point model
- it adopts a weaving model
- it provides some nameable abstraction to modularize crosscutting concerns
  - it provides some means of specifying join points (crosscutting interfaces)
  - it provides some means of specifying enhancements to be combined at the join points (crosscutting features)
- it supports the aspect-base dichotomy
- it supports some sort of quantification over the structure or the behavior of components.
- it assumes that components are oblivious about aspects.

If we abstract out the component model (since the join point model already depends on it) and the weaving model, the above definition can be summarized with the following equation for "aspect-orientedness":

$$\begin{aligned} \text{aspect-oriented} \quad = \quad & \text{aspects} + \text{join point model} + \text{crosscutting} + \\ & \text{obliviousness} + \text{quantification} + \text{aspect-base dichotomy} \end{aligned} \tag{1}$$

Discussion. Aspect-oriented languages can be *domain-specific languages* or *general-purpose languages*. In both cases, they should provide an interpretation for the elements of the *aspect model.*

Example. AspectJ [15] and AspectC [8] are general-purpose aspect-oriented programming languages since their conceptual framework is the *aspect model.* Nevertheless, AspectJ adopts the object model as its component model (Java as the component language) while AspectC adopts the procedural model (C as the component language).

3.2 Assessing Language Features

In this Section, we use the *aspect model* to review four ASoC approaches – AspectJ, Hyper/J, Composition Filters and Demeter/DJ – in order to assess whether they are aspect-oriented or not. Table 3 presents a summary of the resulting interpretations.

	AspectJ	Hyper/J	Composition Filters	DJ
Component Model	object model	object model	object model	object model
Component Component language	object Java	object Java	object object-oriented	object Java
Join Point Model				
Dynamic JPs Static JPs	points in execution classes	. . . classes, fields, methods	sending and receiving messages . . .	nodes in call graph nodes in class graph
Core Model				
Aspect Crosscutting interface Crosscutting feature	aspect pointcuts introduction, advice	hyperslice? hypermodule fields, methods	filter filter expressions wrappers	adaptive method traversal strategies adaptive visitor
Weaving Model	static	static	static/dynamic	static
Aspect-base dichotomy	yes	no	yes	yes
Quantification	yes	yes	yes	yes
Obliviousness	yes	yes	yes	yes

Table 3: Interpretations using the Aspect Model

3.2.1 AspectJ

AspectJ [15] is an aspect-oriented extension to Java that supports general-purpose aspect-oriented programming. AspectJ was developed by a Xerox PARC team that actually proposed the AOP paradigmatic ideas and most of the AOP terminology [16]. Last year, AspectJ was transferred from PARC to an openly-developed eclipse.org project [3].

Terminology. Aspect-oriented programming, aspect, join point, crosscutting, weaving, pointcut, advice, inter-type declaration (introduction).

3.2.2 DJ

The DJ (Demeter/Java) library [19] is a Java package that supports Adaptive Programming (AP) [18], one of the first approaches to ASoC. AP supports the separation of behavioral concerns (methods, strategies) from structural concerns (class graph), in the context of object-oriented software.

Terminology. Adaptive programming, class graph, adaptive method, traversal strategy, adaptive visitor.

Discussion. AP is considered a special case of AOP, where components are expressible in terms of graphs and aspects (*adaptive methods*) refer to and affect the graphs using *traversal strategies* and *adaptive visitors*. According to our conceptual framework, the DJ library supports AOP.

3.2.3 Composition Filters

The Composition Filters (CFs) approach [2, 4] is a modular and orthogonal extension to the object model to cope with object-oriented modeling problems and to increase adaptability and reusability in object-oriented systems. In the CF approach, aspects are expressed as *filters*. A *filter* is defined as a function that manipulates messages received and sent by objects [4]. Filters are language independent.

Terminology. Composition Filters, input filter, output filter, filter type, filter interface, filter element, internal class, condition, selector, superimposition, dispatch filter.

Discussion. CFs is considered a special case of AOP, where filters are wrapped around base components to provide the crosscutting behavior. According to our conceptual framework, the CFs approach supports AOP.

3.2.4 Hyper/J

Hyper/J [13] is a tool developed at IBM Watson Research Center to support Multi-Dimensional Separation of Concerns (MDSoC) [31], an evolution on early work on Subject-Oriented Programming [12]. Hyper/J allows the modularization and composition of concerns without requiring special language extensions. Hyper/J uses *hyperslices* – sets of Java packages, classes, methods, fields, etc. – to modularize any kind of concern.

Terminology. Multi-dimensional separation of concerns, hyperspace, hyperslice, hypermodule, concern map, corresponding unit, merge, override.

Discussion. Hyper/J allows the composition of multiple, separate object models; it does not require a distinguished base and it *does not* support the aspect-base dichotomy. Moreover, hyperslices are *unable* to modularize crosscutting concerns, that is, they do not localize both crosscutting interfaces *and* crosscutting features. Therefore, according to our conceptual framework, *Hyper/J does not support AOP*.

3.3 Supporting Language Design

The *aspect model* subsumes essential concepts and properties for supporting the design of aspect-oriented languages. These essential concepts can be viewed as candidate *higher level abstractions* for aspect-oriented languages.

As a conceptual *framework*, the *aspect model* needs to be instantiated in order to be used. This means that the designer of a new language must: (i) adopt a component model, (ii) adopt a suitable joint point model, (iii) adopt a weaving model, (iv) provide suitable representations for the elements of the adopted models, (v) provide a clear semantics for crosscutting, (vi) provide some means to support quantification, among other things.

The previous list of design tasks is not complete; a *cookbook* for using the *aspect model* to support language design will be the subject of future work.

Example. We are developing `aSide`, *a modeling language for specifying and communicating aspect-oriented designs* that supports the *aspect model.* Some of the major decisions concerning the design of `aSide` are [6]:

- The component model is the UML [1] object model, that is, the UML metamodel.
- The join point model includes *static join points*, comprising some elements used in UML structural models, and *dynamic join points*, comprising some elements used in UML behavioral models.
- The weaving model is static. Weaving diagrams are provided in order to present a clear higher level description of the system after weaving.
- The core model includes: (i) modeling elements that correspond to its core concepts – *aspect* (a parameterized modeling element) and *crosscutting* (a relationship) – and, (ii) modeling elements that correspond to auxiliary concepts such as, *crosscutting interface*, *crosscutting feature*, etc.
- The core model is extended to provide *relationships among aspects* (precedence, requirement, exclusion, etc.) that become evident after the component and join point models are defined.

A detailed description of the aSide modeling language will be the subject of future work.

4 Conclusions

In this paper, we present an *aspect model* for AOSD, a software engineering paradigm that emphasizes the principles of separation of concerns and modularity, the modularization of crosscutting concerns and the properties of quantification and obliviousness. The proposed *aspect model* stands for a theory of aspects, that is, a conceptual framework for AOP that provides consistent terminology and basic semantics for thinking about a problem in terms of the *concepts* and *properties* that characterize the aspect-oriented style as an emerging paradigm to software development.

Furthermore, we use the *aspect model* to present a definition for AOLs and also a characterization of some representative approaches to advanced separation of concerns using our *aspect model.*

4.1 Related Work

Masuhara and Kiczales are seeking to find common frameworks for building models of AOSD mechanisms. In [21], they provide a framework to model the core semantics of five aspect-oriented software development technologies. They try to characterize which properties of a mechanism enable crosscutting modularity, as opposed to hierarchical and block structured modularity. One critical property of their framework is that it models the join points as existing in the result of the weaving process rather than being in either of the input programs.

At Lancaster, while discussing standard interface support for runtime inspection of aspect-oriented programs [22], Mehner and Rashid argue that such a standard interface should be grounded in a common foundation for AOP. In [23] they present the current state of *GEMA*, their generic model for AOP.

Nagy, Aksit and Bergmans argue that the aspect composition mechanisms are an important characteristic of AOLs; in [25] they present *Composition Graphs* (CG), a generic model that allows the uniform description and comparison of different aspect-oriented composition mechanisms.

This work as well as the other mentioned above are just initial, short steps towards the goal of an unifying conceptual framework for AOSD; as an interesting remark, all of them are looking specifically at programming approaches as these are relatively well-established compared to design or requirement-level approaches.

4.2 Ongoing Work

The adoption of an unifying conceptual framework for AOP is an important step for characterizing the design space of aspect-oriented languages and providing support for AOSD. The definition of the *aspect model* is our first, small step in that direction. In this context, our ongoing work includes the following additional steps:

- The provision of a formal semantics for the *aspect model.* The definition of a sound composition semantics for *crosscutting* is important to clarify the interplay between aspects and components, and to manage the possible interferences between *crosscutting* and conventional composition mechanims.

- The complete specification of the `aSide` modeling language in order to provide notation and rules that enable the creation of structural and behavioral models in which aspects are explicitly treated as first-class citizens.

- The use of the *aspect model* to develop a case tool that supports: (i) aspect-oriented modeling with aSide, (ii) a metrics suite for assessing aspect-oriented software, and (iii) code generation from aSide models.

Acknowledgements.

We would like to thank the anonymous referees for making several suggestions that have improved our paper. The authors are supported by the PRONEX Project under grant 7697102900, by ESSMA under grant 552068/2002-0 and by the Art. 1st of Decree number 3.800, of 04.20.2001. This work is dedicated to the memory of Prof. Sergio Carvalho.

References

[1] Unified Modeling Language (UML) Specification, Version 1.4, 2002. http://www.omg.org/uml/.

[2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Workshop on Object-Based Distributed Programming at ECOOP'93*, pages 152–184. Springer-Verlag, 1993.

[3] AspectJ project, 2003. http://www.eclipse.org/aspectj/.

[4] L. Bergmans and M. Aksit. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[5] G. Booch. *Object-Oriented Design with Applications*. Benjamin-Cummings, 1991.

[6] C. Chavez and C. Lucena. A Metamodel for Aspect-Oriented Modeling. In *Workshop on Aspect-Oriented Modeling with the UML at AOSD'02*, April 2002.

[7] P. Chen. The Entity Relationship Model - Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[8] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-specific Customization in Operating System Code. In *Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering (FSE-9)*, 2001.

[9] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[10] R. Filman. What Is Aspect-Oriented Programming, Revisited. In *Workshop on Advanced Separation of Concerns at ECOOP'01*, June 2001.

[11] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Int'l Workshop on Advanced Separation of Concerns at OOPSLA'00*, 2000.

[12] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *7th Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA93)*, pages 411–428, 1993.

[13] Hyper/J Web Page, 2001. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm.

[14] G. Kiczales. Aspect-Oriented Programming. *ACM Computing Surveys*, 28(4es):154, 1996.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *11th Eur. Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[17] J. Lamping. The Role of Base in Aspect-oriented Programming. In *Int'l Workshop on Aspect-Oriented Programming at ECOOP'99*, 1999.

[18] K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[19] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, 44(10):39–41, October 2001.

[20] C. Lopes and G. Kiczales. D: A Language Framework for Distributed Programming. Technical Report SPL-97-010, Palo Alto Research Center, 1997.

[21] H. Masuhara and G. Kiczales. A Framework for Modeling Aspect-Oriented Mechanisms, 2003. Revised version to appear in ECOOP'03.

[22] K. Mehner and A. Rashid. Towards a Standard Interface for Runtime Inspection in AOP Environments. In M. Chu-Carrol and G. Murphy, editors, *Workshop on Tools for Aspect-Oriented Software Development at OOPSLA'02*, 2002.

[23] K. Mehner and A. Rashid. GEMA: A Generic Model for AOP (Extended Abstract). In *Belgian and Dutch Workshop on Aspect-Oriented Programming*, 2003.

[24] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A Case-Study for Aspect-Oriented Programming. Technical Report SPL-97-009, Palo Alto Research Center, 1997.

[25] I. Nagy, M. Aksit, and L. Bergmans. Composition Graphs: a Foundation for Reasoning about Aspect-Oriented Composition. In *Workshop on Foundations of Aspect-oriented Languages (FOAL) at AOSD'2003*, 2003.

[26] H. Ossher and P. Tarr. Operation-Level Composition: A Case in (Join) Point. In *Int'l Workshop on Aspect-Oriented Programming at ECOOP'98*, 1998.

[27] H. Ossher and P. Tarr. Using Multi-dimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, October 2001.

[28] K. Ostermann and G. Kniesel. Independent Extensibility—An Open Challange for AspectJ and Hyper/J. In *Int'l Work. on Aspects and Dimensional Computing at ECOOP'00*, 2000.

[29] A. Ryman. The Theory-Model Paradigm in Software Design. Technical Report TR74.048, IBM Tech. Report, IBM Toronto, Ont., October 1989.

[30] A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[31] P. Tarr, H. Ossher, W. Harrison, and S. S. Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *21st Int'l Conf. on Software Engineering (ICSE'99)*, pages 107–119, May 1999.

[32] P. Wegner. Dimensions of Object-based Language Design. In *2nd Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 168–182, October 1987.